Deep Learning Guide Book

Nishant Baheti

Apr 18, 2024

MATH

CHAPTER ONE

ARCHITECTURE

CHAPTER

TWO

SYMBOLS & NAMING CONVENTIONS

$$
\begin{aligned}
n &= \text{number of nodes} \\
l &= \text{layer number} \\
w, W &= \text{weights matrix} \\
b &= \text{bias matrix} \\
z, Z &= \text{hypthesis result (result before applying activation function)} \\
g(z) &= \text{activation function} \\
a, A &= \text{activation matrix (result after applying activation function)} \\
x, X &= \text{input to network} \\
\hat{y} &= \text{output of network}
\end{aligned}
$$

values for forward propogation

$$
\begin{aligned}
n^{[l]} &= \text{number of nodes in the layer} \\
z^{[l]} &= \text{hypothesis result of the layer} \\
w^{[l]} &= \text{weights results of the layer} \\
b^{[l]} &= \text{bias results of the layer} \\
a^{[l]} &= \text{activation results of the layer}
\end{aligned}
$$

derivatives for backward propogation

$$
\begin{aligned}
dw`^{[l]} &= \frac{\partial L}{\partial w} \rightarrow \text{loss derivative based on weights} \\
db^{[l]} &= \frac{\partial L}{\partial b} \rightarrow \text{ loss derivative based on biases} \\
dz^{[l]} &= \frac{\partial L}{\partial z} \rightarrow \text{ loss derivative based on hypothesis result} \\
da^{[l]} &= \frac{\partial L}{\partial a} \rightarrow \text{ loss derivative based on activation r`esult}
\end{aligned}
$$

CHAPTER

THREE

FLOW

CHAPTER FOUR

SHAPES

$W^{[l]}$	$(n^{[l]}, n^{[l-1]})$	$dW^{[l]}$
$b^{[l]}$	$(1, n^{[l]})$	$db^{[l]}$
$Z^{[l]}$	$(n^{[l]}, n^{[l-1]})$	$dZ^{[l]}$
$A^{[l]}$	$(n^{[l]}, n^{[l-1]})$	$dA^{[l]}$

where

l = layer number >= 1

$A^{[0]}$ = X

CHAPTER

FIVE

DENSE LAYER

How do we actually initialize a layer for a New Neural Network?

- initialization of weights with small random values
  - why? because according to Andrew Ng's explanation if all the weights/params are initialized by zero or same value then all the hidden units will be symmetric with identical nodes.
  - With identical nodes there will be no learning/ decision making. because all the decisions shares same value.
  - If all the nodes will have zero values(weights are zero , multiplication with weights will also be zero) and propogation result wont be a conclusive one(dead network).
- initialization of bias can be zero.
  - as randomness is already introduced by weights. But for smaller Neural Network it is advised to not to initialize with zero.

$$
\begin{aligned}
X &= \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \dots & x_1^{(m)} \\ x_2^{(1)} & x_2^{(2)} & \dots & x_2^{(m)} \\ & & \vdots & \\ x_n^{(1)} & x_n^{(2)} & \dots & x_n^{(m)} \end{bmatrix}_{n \times m} \\
W &= \begin{bmatrix} w_1^{(1)} & w_1^{(2)} & \dots & w_1^{(m)} \\ w_2^{(1)} & w_2^{(2)} & \dots & w_2^{(m)} \\ & & \vdots & \\ w_n^{(1)} & w_n^{(2)} & \dots & w_n^{(m)} \end{bmatrix}_{n \times m} \\
b &= \begin{bmatrix} b_1 & b_2 & \dots & b_n \end{bmatrix}_{1 \times n} \\
Z &= XW^T + b \\
&= \begin{bmatrix} x_1^{(1)} & x_1^{(2)} & \dots & x_1^{(m)} \\ x_2^{(1)} & x_2^{(2)} & \dots & x_2^{(m)} \\ & & \vdots & \\ x_n^{(1)} & x_n^{(2)} & \dots & x_n^{(m)} \end{bmatrix}_{n \times m} \begin{bmatrix} w_1^{(1)} & w_2^{(1)} & \dots & w_n^{(1)} \\ w_1^{(2)} & w_2^{(2)} & \dots & w_n^{(2)} \\ & & \vdots & \\ w_1^{(m)} & w_2^{(m)} & \dots & w_n^{(m)} \end{bmatrix}_{m \times n} + \begin{bmatrix} b_1 & b_2 & \dots & b_n \end{bmatrix}_{1 \times n} \\
&= \begin{bmatrix} x_1^{(1)} w_1^{(1)} + x_1^{(2)} w_1^{(2)} + \dots + x_1^{(m)} w_1^{(m)} & \dots & x_1^{(1)} w_n^{(1)} + x_1^{(2)} w_n^{(2)} + \dots + x_1^{(m)} w_n^{(m)} \\ x_2^{(1)} w_1^{(1)} + x_2^{(2)} w_1^{(2)} + \dots + x_2^{(m)} w_1^{(m)} & \dots & x_2^{(1)} w_n^{(1)} + x_2^{(2)} w_n^{(2)} + \dots + x_2^{(m)} w_n^{(m)} \\ & \vdots & \\ x_n^{(1)} w_1^{(1)} + x_n^{(2)} w_1^{(2)} + \dots + x_n^{(m)} w_1^{(m)} & \dots & x_n^{(1)} w_n^{(1)} + x_n^{(2)} w_n^{(2)} + \dots + x_n^{(m)} w_n^{(m)} \end{bmatrix}_{n \times n} + \begin{bmatrix} b_1 & b_2 & \dots & b_n \\ b_1 & b_2 & \dots & b_n \\ & & \vdots & \\ b_1 & b_2 & \dots & b_n \end{bmatrix}_{n \times n \text{ broadcast}} \\
&= \begin{bmatrix} x_1^{(1)} w_1^{(1)} + x_1^{(2)} w_1^{(2)} + \dots + x_1^{(m)} w_1^{(m)} + b_1 & \dots & x_1^{(1)} w_n^{(1)} + x_1^{(2)} w_n^{(2)} + \dots + x_1^{(m)} w_n^{(m)} + b_n \\ x_2^{(1)} w_1^{(1)} + x_2^{(2)} w_1^{(2)} + \dots + x_2^{(m)} w_1^{(m)} + b_1 & \dots & x_2^{(1)} w_n^{(1)} + x_2^{(2)} w_n^{(2)} + \dots + x_2^{(m)} w_n^{(m)} + b_n \\ & \vdots & \\ x_n^{(1)} w_1^{(1)} + x_n^{(2)} w_1^{(2)} + \dots + x_n^{(m)} w_1^{(m)} + b_1 & \dots & x_n^{(1)} w_n^{(1)} + x_n^{(2)} w_n^{(2)} + \dots + x_n^{(m)} w_n^{(m)} + b_n \end{bmatrix}_{n \times n}
\end{aligned}
$$

5.1 Forward

$$
\begin{aligned}
Z^{[1]} &= A^{[0]} W^{[1]T} + b^{[1]} \\
A^{[1]} &= g^{[1]}(Z^{[1]}) \\
\\
Z^{[2]} &= A^{[1]} W^{[2]T} + b^{[2]} \\
A^{[2]} &= g^{[2]}(Z^{[2]})
\end{aligned}
$$

`Generalized`

$$
\begin{aligned}
Z^{[l]} &= A^{[l-1]} W^{[l]T} + b^{[l]} \\
A^{[l]} &= g^{[l]}(Z^{[l]})
\end{aligned}
$$

[1]:
```
from abc import ABC,abstractmethod
import numpy as np
import matplotlib.pyplot as plt
```

lets take two layers

- lets take layer 1 as input layer. This means input is x or $a^{[0]}$
  - lets take 3 columns = number of nodes = $n^{[0]} = 3$
  - and take 10 samples = m = 10
  - shape of $a^{[0]} = (n^{[0]}, m)$ (3, 10)
  - shape of $w^{[1]} = (n^{[0]}, m) = dw^{[1]}$ (3, 10)
  - shape of $b^{[1]} = (1, n^{[0]}) = db^{[1]}$ (1, 3)
  - shape of $z^{[1]} = (n^{[0]}, m)(m, n^{[0]}) + (1, n^{[0]}) = (n^{[0]}, n^{[0]}) = dz^{[1]}$ (3, 10) (10, 3)+ (1, 3) = (3, 3)
  - shape of $z^{[1]}$ = shape of $a^{[1]} = (n^{[0]}, n^{[0]})$ (3, 3)
- lets take layer 2 the next layer to that. The first one in hidden layer. Input to this layer is $a^{[1]}$
  - lets take number of nodes in the layer = 5 = $n^{[1]} = 5$
  - shape of $w^{[2]} = (n^{[1]}, n^{[0]}) = dw^{[2]}$ (5 ,3)
  - shape of $b^{[2]} = (1, n^{[1]}) = db^{[2]}$ (1, 5)
  - shape of $z^{[2]} = (n^{[0]}, n^{[0]})(n^{[0]}, n^{[1]}) + (1, n^{[1]}) = (n^{[0]}, n^{[1]}) = dz^{[2]}$ (3, 3) (3, 5) + (1, 5) = (3, 5)

[2]:
```
n0 = 3
n1 = 5
m = 10
```

[3]:
```
a0 = np.random.random((n0, m))
w1 = np.random.random((n0, m))
b1 = np.random.random((1, n0))
print(w1.shape, a0.shape,'+', b1.shape)
```

```
(3, 10) (3, 10) + (1, 3)
```

[4]:
```
z1 = (a0 @ w1.T) + b1
z1.shape
```

[4]:
```
(3, 3)
```

[5]:
```
a1 = 1/(1 + np.exp(-z1))

a1.shape
```

[5]:
```
(3, 3)
```

[6]:
```
w2 = np.random.random((n1, n0))
b2 = np.random.random((1, n1))
print(w2.shape, a1.shape,'+', b2.shape)
```

```
(5, 3) (3, 3) + (1, 5)
```

[7]:
```
z2 = (a1 @ w2.T) + b2
z2.shape
```

[7]:
```
(3, 5)
```

[8]:
```
a2 = 1/(1 + np.exp(-z2))
a2.shape
```

[8]:
```
(3, 5)
```

5.2 Backward

param for this layer (this function starts working from here)

$$dW = dZ'.A^T$$

$$dB = \sum(dZ')$$

input for next layer (in backward propogation)

$$dZ = dZ'.W^T$$

[9]:
```
dz2 = np.random.random((n0,n1))
dz2.shape
```

[9]:
```
(3, 5)
```

[10]:
```
dw2 = dz2 @ a2.T
dw2.shape
```

[10]:
```
(3, 3)
```

[11]:
```
db2 = dz2.sum(axis=0,keepdims=True)
db2.shape
```

[11]:
```
(1, 5)
```

[12]:
```
dz1 = dz2 @ w2
dz1.shape
```

[12]:
```
(3, 3)
```

[13]:
```
dw1 = dz1 @ a1.T
dw1.shape
```

[13]:
```
(3, 3)
```

[14]:
```
db1 = dz1.sum(axis=0,keepdims=True)
db1.shape
```

[14]:
```
(1, 3)
```

[15]:
```
dz1 @ w1
```

[15]:
```
array([[1.00430696, 1.12665459, 1.27528356, 0.37028909, 1.83008842,
         0.86290497, 1.23745471, 1.23044548, 0.83923269, 1.65279249],
        [0.77372465, 0.99710549, 1.00752794, 0.2431575 , 1.27532378,
         0.7486004 , 1.11498651, 0.84140139, 0.61524338, 1.5975826 ],
        [1.57524514, 1.82300126, 2.04126706, 0.56541619, 2.87108659,
         1.40560442, 2.03900305, 1.88715055, 1.31532754, 2.74793296]])
```

5.3 Model

[16]:
```
class LayerDense:
    """Layer Module

    It is recommended that input data X is scaled(data scaling operations)
    so that data is normalized but meaning of the data remains same.

    Args:
        n_inputs (int) : number of inputs
        n_neurons (int) : number of neurons
    """
    def __init__(self,n_inputs,n_neurons):
        """
        """
        self.w = 0.10 * np.random.randn(n_inputs,n_neurons) # multiply by 0.1 to make it␣
↪small
        self.b = np.zeros((1,n_neurons))

    def forward(self, a):
        """forward propogation calculation
        """
        self.a = a
        self.z = np.dot(self.a,self.w)+self.b

    def backward(self, dz):
        """backward pass
        """
        # gradient on parameters
        self.dw = dz @ self.a.T
        self.db = dz.sum(axis=0,keepdims=True)

        # gradient on values / input to next layer in backpropogation
        self.dz = dz @ self.w
```

CHAPTER

SIX

ACTIVATION FUNCTIONS

Notes:

1. Introducing non linearity to the network. Why?
2. According to me we need one parameter to compare all the nodes results after learning and passing the value to upcoming nodes.
3. To make sense of the data and a mapping for approximation.
4. Understand what is the impact of weights and biases changing value to the network/nodes. If there is only linear fx then it can only fit linear data but if we have not linear data like a sine wave then it will fail to do so.
5. If there is no activate function then the whole network will be similar to a one linear node.

$w^T(w^T(w^Tx + b) + b) + b... = output$

[3]:
```
import numpy as np
import matplotlib.pyplot as plt
```

6.1 Sigmoid

$f(x) = \frac{1}{(1+e^{-x})}$

- granular
- between 0 and 1
- Comparatively complex calcultaion

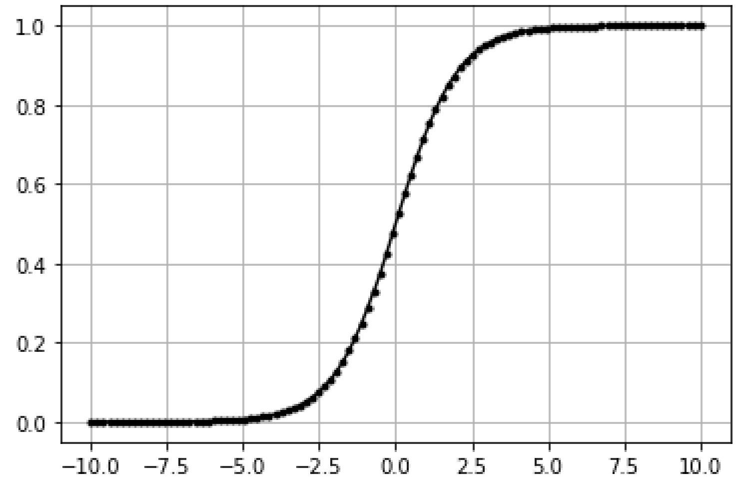
[17]:
```
class ActivationSigmoid:
    """Sigmoid Activation Fx
    """
    def forward(self, inputs):
        """Apply Sigmoid to input
        """
        self.output = 1 / (1 + np.exp(-inputs))
```

[18]:
```
data = np.linspace(-10, 10, 100)
act = ActivationSigmoid()
act.forward(data)
```

(continues on next page)

(continued from previous page)

```
plt.plot(data,act.output,'k.-')
plt.grid()
```



6.2 Stepwise

$f(x) = 0$ | if $x \leq 0$

$f(x) = 1$ | if $x0$

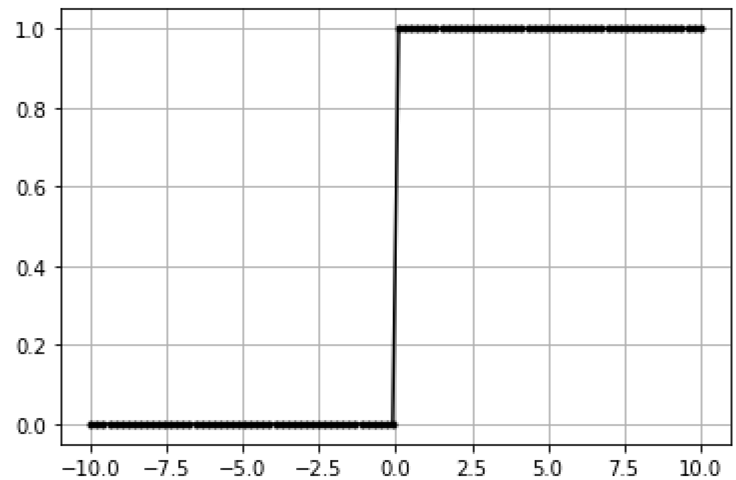
- non granular
- only 0 and 1

[19]:

```
class ActivationStepwise:
    """Stepwise Activation Fx
    """
    def forward(self, inputs):
        """Apply Stepwise to inputs

        Args:
            inputs (numpy.ndarray) : input matrix
        """
        self.inputs = inputs # save inputs
        self.output = (inputs > 0).astype('int') # calculate from inputs
```

[20]:

```
data = np.linspace(-10, 10, 100)
act = ActivationStepwise()
act.forward(data)


plt.plot(data,act.output,'k.-')
plt.grid()
```



6.3 Relu

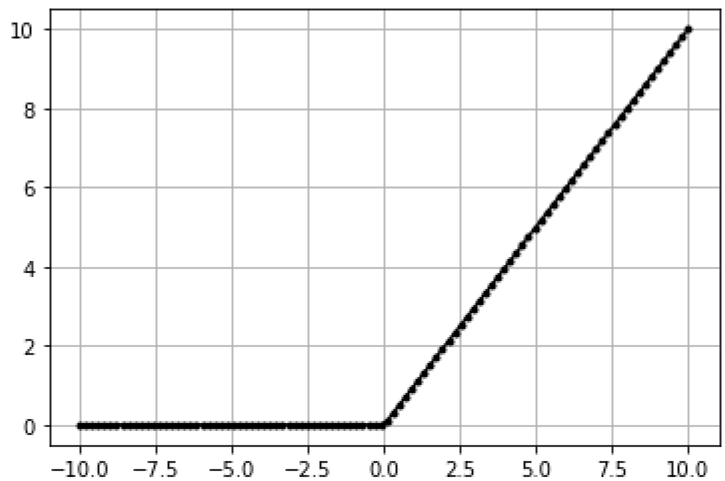
$f(x) = 0$ | if $x \leq 0$

$f(x) = x$ | if $x0$

- granular
- between 0 to x
- easy calculation
- almost linear but rectified so less than zeros are not allowed.so introducing slight non linearity makes it eligible for an activation function but also inherently easy and fast calculation than sigmoid.

[21]:
```
class ActivationReLU:
    """ReLU Activation Fx
    """

    def forward(self, inputs):
        """Apply ReLU to input

        Args:
            inputs (numpy.ndarray) : input matrix
        """
        self.inputs = inputs # save inputs
        self.output = np.maximum(0, inputs) # calculate from inputs

    def backward(self, dvalues):
        """Apply backward propogation

        Args:
            dvalues (numpy.ndarray) : inputs from previous later in backward prop
        """
        self.dinputs = dvalues.copy()
        self.dinputs[self.inputs <= 0] = 0
```

[22]:
```
data = np.linspace(-10, 10, 100)
act = ActivationReLU()
act.forward(data)


plt.plot(data,act.output,'k.-')
plt.grid()
```



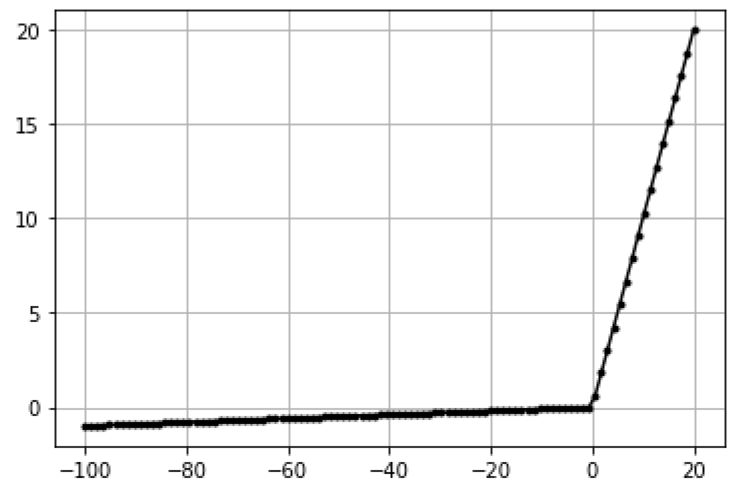
6.4 Leaky Relu

$f(x) = 0.01x$ | if $x \leq 0$

$f(x) = x$ | if $x 0$

[34]:
```
class ActivationLeakyReLU:
    """ReLU Activation Fx
    """

    def forward(self, inputs):
        """Apply Leaky ReLU to input

        Args:
            inputs (numpy.ndarray) : input matrix
        """
        self.inputs = inputs # save inputs
        self.output = np.where(inputs > 0,inputs,0.01 * inputs)
```

[40]:
```
data = np.linspace(-100, 20, 100)
act = ActivationLeakyReLU()
act.forward(data)


plt.plot(data,act.output,'k.-')
plt.grid()
```



6.5 Softplus

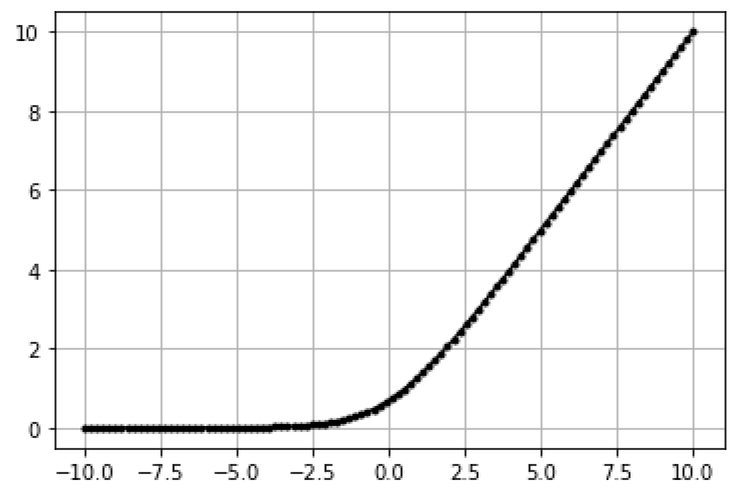
smooth ReLU function

:nbsphinx-math:`begin{align}
f(x) &= log{(1 + exp(x))}

end{align}`

[1]:
```
class Softplus:
    def forward(self, inputs):
        """Apply Leaky ReLU to input

        Args:
            inputs (numpy.ndarray) : input matrix
        """
        self.inputs = inputs # save inputs
        self.output = np.log(1 + np.exp(self.inputs))
```

[4]:
```
data = np.linspace(-10, 10, 100)
act = Softplus()
act.forward(data)


plt.plot(data,act.output,'k.-')
plt.grid()
```



6.6 Hyperbolic Tangent(Tanh)

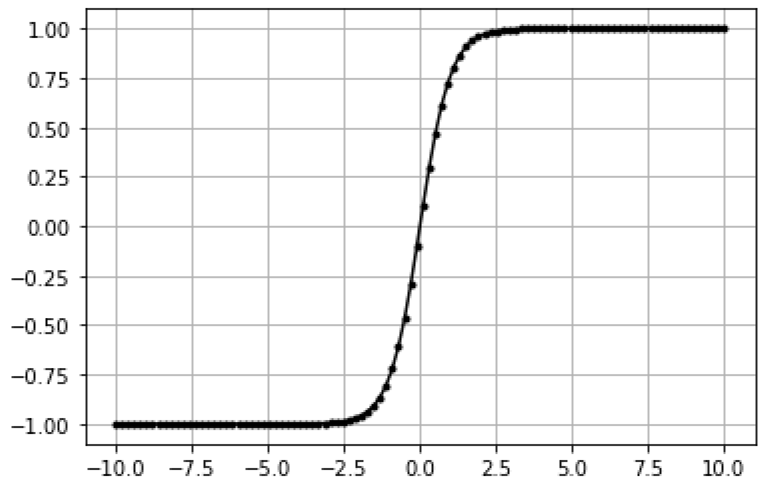
:nbsphinx-math:`begin{align}

f(x) &= Tanh(x)
&= frac{2}{1 + e^{-2x}} - 1\\ &= frac{1 - e^{-2x}}{1 + e^{-2x}}\\ &= frac{e^x - e^{-x}}{e^x + e^{-x}}

end{align}`

[48]:
```
class ActivationTanh:

    def forward(self, inputs):
        """Apply Leaky ReLU to input

        Args:
            inputs (numpy.ndarray) : input matrix
        """
        self.inputs = inputs # save inputs

#         # scratch
#         ez = np.exp(inputs)
#         e_z = np.exp(-inputs)
#         self.output = (ez - e_z)/(ez + e_z)

        self.output = np.tanh(inputs)
```

[51]:
```
data = np.linspace(-10, 10, 100)
act = ActivationTanh()
act.forward(data)


plt.plot(data,act.output,'k.-')
plt.grid()
```



6.7 Softmax

:nbsphinx-math:`begin{align}

sigma(mathbf{z})_i &= frac{e^{z_i}}{sum_{j=1}^m e^{z_j}} \ \ & text{ for } i = 1, dotsc , m text{ and } mathbf z =(z_1,dotsc,z_m) in R^m\ \ sigma &= text{softmax}\ vec{z} &= text{input vector}\ e^{z_{i}} &= text{standard exponential function for input vector}\ K &= text{number of classes in the multi-class classifier}\ e^{z_{j}} &= text{standard exponential function for output vector}\

end{align}`

here z is actually z = x - x.max(). because exponential values increase really fast. and that can cause out of memory error. so we can't use x directly.

when x - x.max() is done then the largest value is 0. so values will not blow out.

[25]:
```
np.exp(1000) # like this
```

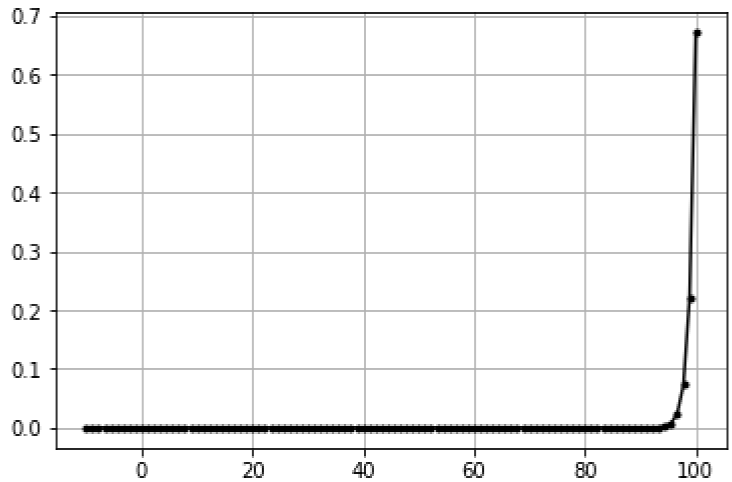
```
<ipython-input-25-87fe7bad57ec>:1: RuntimeWarning: overflow encountered in exp
  np.exp(1000) # like this
```

[25]:
```
inf
```

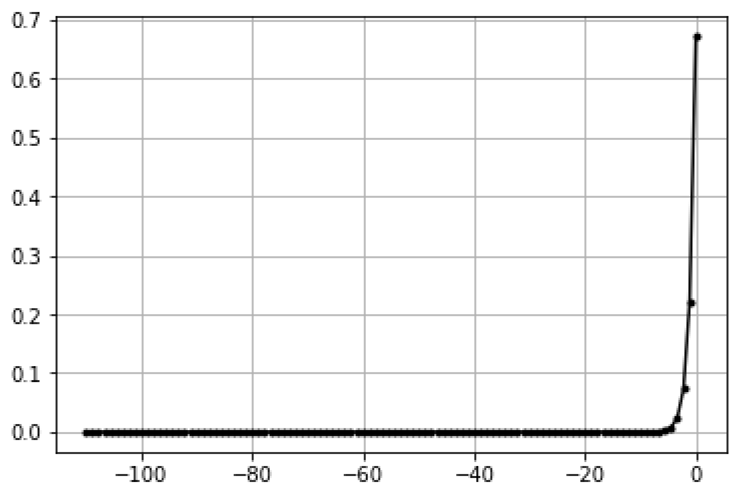
[26]:
```
class ActivationSoftmax:

    def forward(self, inputs):
        """Forward propogation calculation

        Args:
            inputs (numpy.ndarray) : input matrix
        """
        exp_values = np.exp(inputs - inputs.max(axis=1, keepdims=True))
        probabilites = exp_values / exp_values.sum(axis=1, keepdims=True)
        self.output = probabilites
```

[27]:
```
data = np.linspace(-10,100,100).reshape(1,100) #(1,100)
```

[28]:
```
t_exp = np.exp(data)
t_prob = t_exp / np.sum(t_exp, axis=1, keepdims=True)
plt.plot(data[0],t_prob[0],'k.-')
plt.grid()
```



[29]:
```
act = ActivationSoftmax()
act.forward(data)

plt.plot((data - data.max(axis=1, keepdims=True))[0],act.output[0],'k.-')
plt.grid()
```



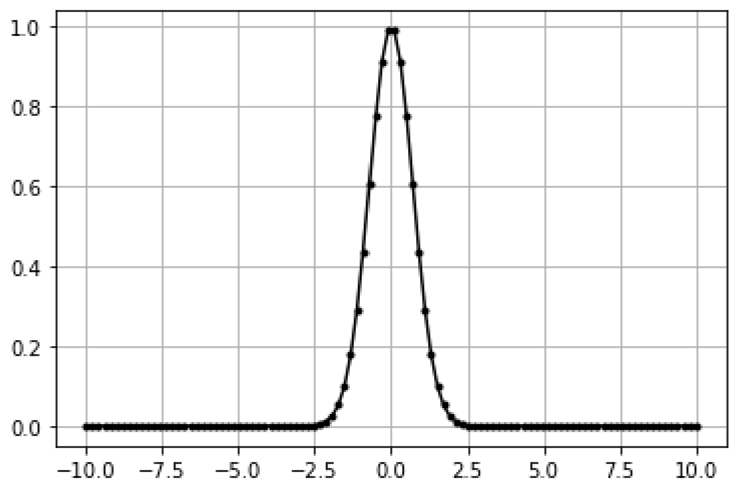
6.8 Gaussian

:nbsphinx-math:`begin{align}
f(x) = exp(-x^2)

end{align}`

[10]:
```
class ActivationGaussian:

    def forward(self, inputs):
        """Forward propogation calculation

        Args:
            inputs (numpy.ndarray) : input matrix
        """
        self.output = np.exp(-(inputs**2))
```

[11]:
```
data = np.linspace(-10, 10, 100)
act = ActivationGaussian()
act.forward(data)

plt.plot(data,act.output,'k.-')
plt.grid()
```



CHAPTER

SEVEN

LOSS

[30]:
```
class Loss(ABC):
    """Loss Meta class
    """
    @abstractmethod
    def __init__(self):
        pass

    @abstractmethod
    def forward(self):
        """mandatory method for child class
        """
        pass

    def calculate(self, output, y):
        """Calculate mean loss

        Args:
            output : output from the layer
            y : truth value/ target/ expected outcome
        """
        # it can be individual outcome of different kind of loss functions
        sample_losses = self.forward(output, y)

        # calculating mean
        data_loss = np.mean(sample_losses)
        return data_loss
```

7.1 Categorical Cross Entropy

$$L(a^{[l]}, y) = ylog(a^{[l]}) + (1 - y)log(1 - a^{[l]})$$

derivative of loss over a –> da

$$\frac{\partial L}{\partial a} = [\frac{y}{a} + \frac{1-y}{1-a}(-1)]$$
$$\frac{\partial L}{\partial a} = [\frac{y}{a} - \frac{1-y}{1-a}]$$

derivative of loss over z –> dz

$$\frac{\partial L}{\partial z} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z}$$

derivative of loss over w –> dw

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial w}$$

derivative of loss over b –> db

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial z}\frac{\partial z}{\partial b}$$

- y_pred_clipped
  - numpy.clip is used to clip the values from min and max values like bandpass filter
  - min = 1.0 * 10^-7
  - max = 1 - 1.0 * 10^-7
- correct_confidences
  - probabilities for target value that has been
  - calculated earlier
  - only for categorical variables

[31]:
```
class LossCategoricalCrossEntropy(Loss):
    """Categorical Cross entropy loss
    """

    def forward(self, y_pred, y_true):
        """forward propogation calculation

        Args:
            y_pred (numpy.ndarray) : predictions generated
            y_true (numpy.ndarray) : actual values
        """

        # get total number of rows/samples
        samples = len(y_pred)
```

(continues on next page)

(continued from previous page)

```
        y_pred_clipped = np.clip(y_pred,1e-7,1-1e-7)

        correct_confidences = None
        if len(y_true.shape) == 1:
            correct_confidences = y_pred_clipped[range(samples),y_true]

        elif len(y_true.shape) == 2:
            correct_confidences = np.sum(y_pred_clipped * y_true, axis = 1)

        else:
            pass

        # losses
        negative_log_Likelihoods = -np.log(correct_confidences)
        return negative_log_Likelihoods
```

CHAPTER EIGHT

OPTIMIZATION ALGORTIHMS

`WORK IN PROGRESS`

8.1 Gradient Descent

8.2 Mini Batch Gradient Descent

CHAPTER

NINE

WHAT IS LOG

Solving for x

$e^x = b$

raising the euler's number to what value that it becomes b.

$log_e(b) = x$

[1]:
```python
import numpy as np

b = 5.2

print(np.log(b))

print(np.exp(np.log(b)))
```

```
1.6486586255873816
5.2
```

CHAPTER

TEN

REGRESSION INTUITION

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import make_regression
import seaborn as sns
from mpl_toolkits import mplot3d
```

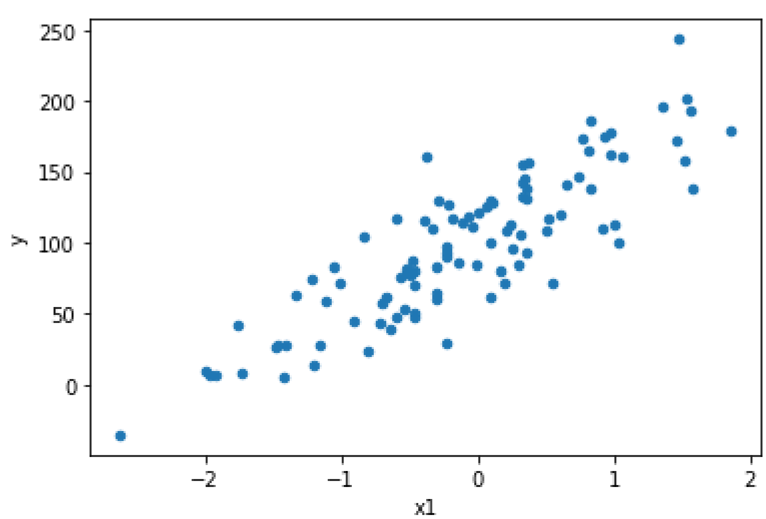
[13]:
```
X, y = make_regression(n_features=1,noise=30,random_state=42,bias=100)
```

[18]:
```
df = pd.DataFrame(np.hstack((X,y.reshape(-1,1))),columns=['x1','y'])
```

[19]:
```
df['x0'] = 1
```

[20]:
```
df.plot(x='x1',y='y',kind='scatter')
```

[20]:
```
<AxesSubplot:xlabel='x1', ylabel='y'>
```



[21]:
```
def plot_regression(x,y,y_hat,figsize=(12,5)):
    fig, ax = plt.subplots(1,2,figsize=figsize)

    ax[0].scatter(x, y, label='original')
    ax[0].plot(x, y_hat, 'k.', label='predicted')
```

(continues on next page)

(continued from previous page)

```
    ax[1].plot(y, label='original')
    ax[1].plot(y_hat, label='predicted')

    plt.legend()
```

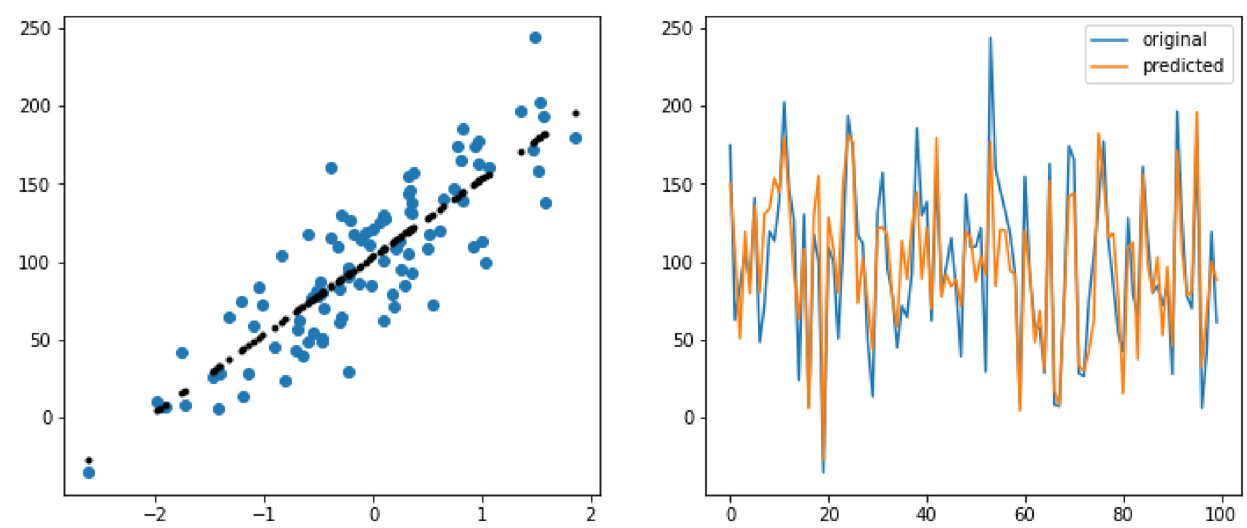
10.1 Fitting a linear regression model

10.1.1 revisiting psuedo inverse

$$X\theta = Y$$
$$\theta = X^{-1}Y$$

[22]:
```
theta = np.linalg.pinv(df[['x0','x1']].values) @ df.y.values
print("theta :",theta)


y_hat = df[['x0','x1']].values @ theta

plot_regression(df.x1,df.y, y_hat)
```

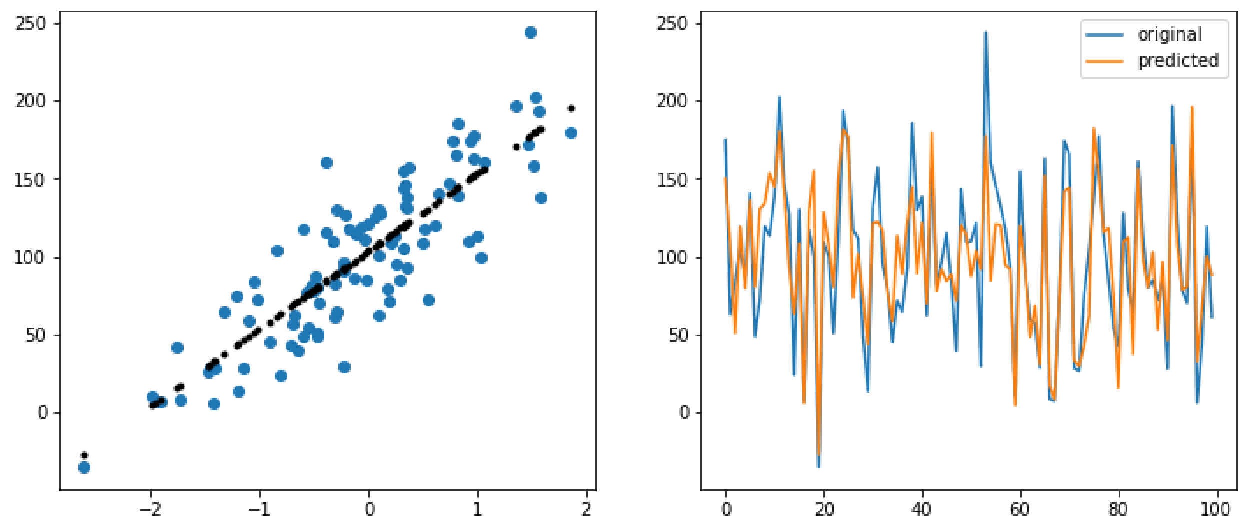
```
theta : [103.49534596  49.82930935]
```



10.1.2 revisiting svd and linear systems

[23]:
```
u,s,vT = np.linalg.svd(df[['x0','x1']].values,full_matrices=False)

theta = vT.T @ np.linalg.pinv(np.diag(s)) @ u.T @ df.y

print("theta :",theta)


y_hat = df[['x0','x1']].values @ theta
plot_regression(df.x1,df.y, y_hat)
```

```
theta : [103.49534596  49.82930935]
```



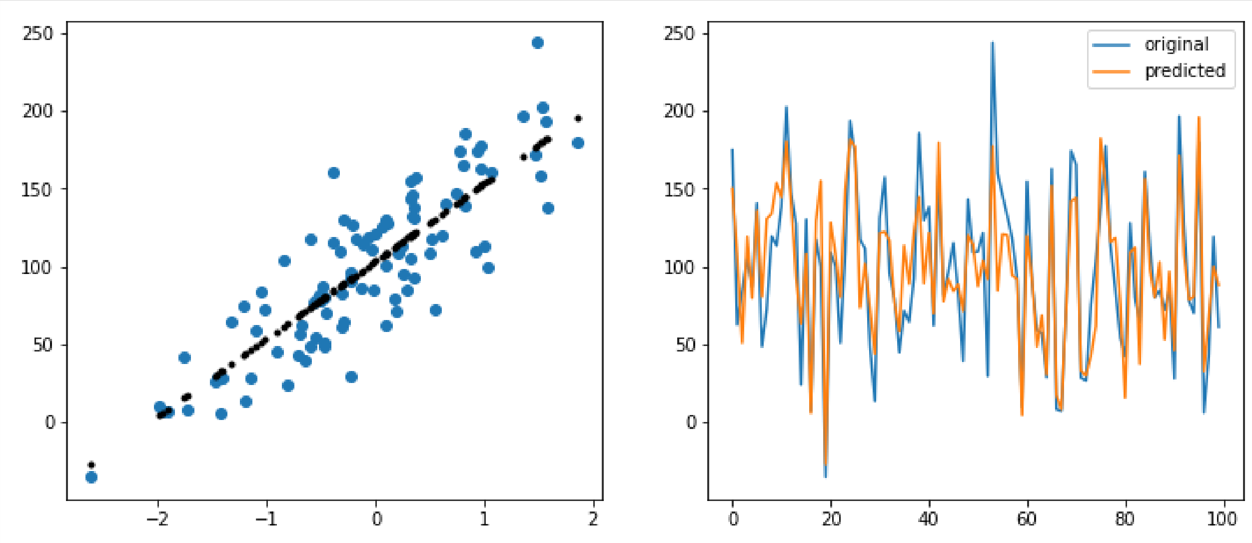
10.1.3 good old sklearn

[24]:
```
from sklearn.linear_model import LinearRegression
```

[25]:
```
model = LinearRegression()
model = model.fit(df[['x0','x1']].values,df.y.values)
```

[26]:
```
y_hat = model.predict(df[['x0','x1']].values)
```

[27]:
```
plot_regression(df.x1,df.y, y_hat)
```



10.2 lets try something with Neural Networks

[28]:
```
import tensorflow as tf
```

10.2.1 A neural net like perceptron

[29]:
```
normalizer = tf.keras.layers.Normalization(axis=-1)
normalizer.adapt(df[['x0','x1']].values) # adapt is like fit
```

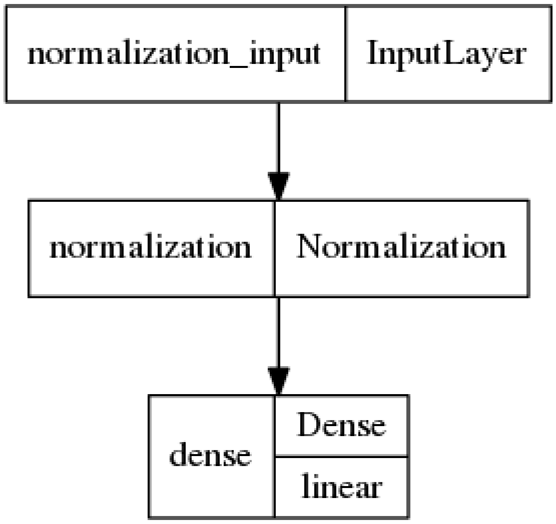
[30]:
```
model = tf.keras.Sequential([
    normalizer,
    tf.keras.layers.Dense(units=1)
])
```

[31]:
```
model.summary()
```

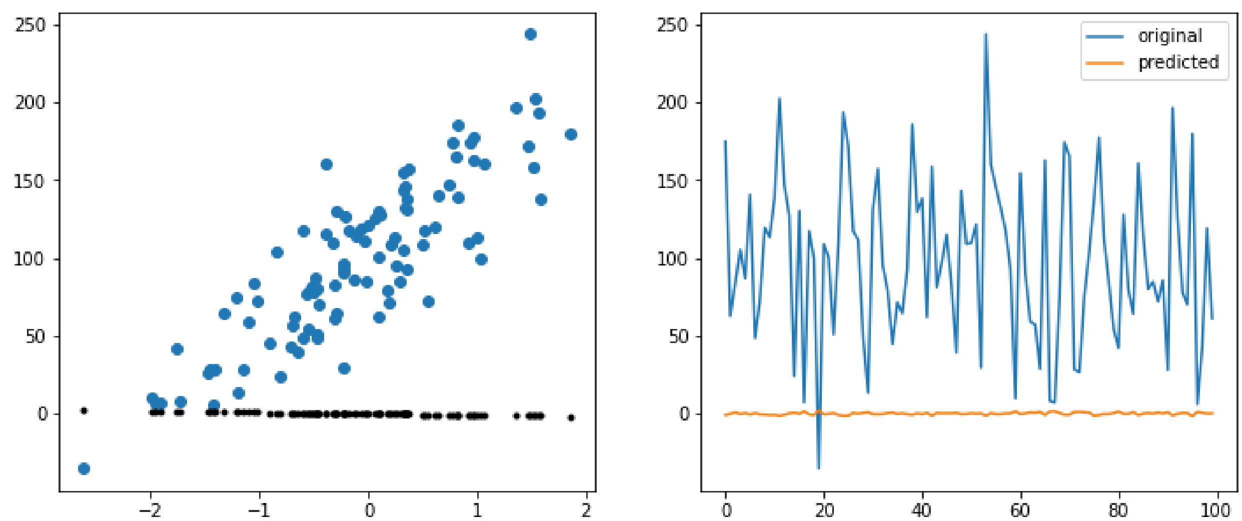
```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 normalization (Normalizatio  (None, 2)                5
 n)

 dense (Dense)               (None, 1)                 3

=================================================================
Total params: 8
Trainable params: 3
Non-trainable params: 5
_________________________________________________________________
```

[32]:
```
tf.keras.utils.plot_model(model,show_layer_activations=True)
```

[32]:



[33]:
```
y_hat = model.predict(df[['x0','x1']].values)

plot_regression(df.x1,df.y,y_hat)
```



it is not trained yet. so result is understandable.

[34]:
```
model.compile(
    optimizer=tf.optimizers.Adam(learning_rate=0.1),
    loss=['mse'],
    metrics=['mse']
)
```

[35]:
```
history = model.fit(
    df[['x0','x1']],
    df.y,
    epochs=1000,
```

(continues on next page)

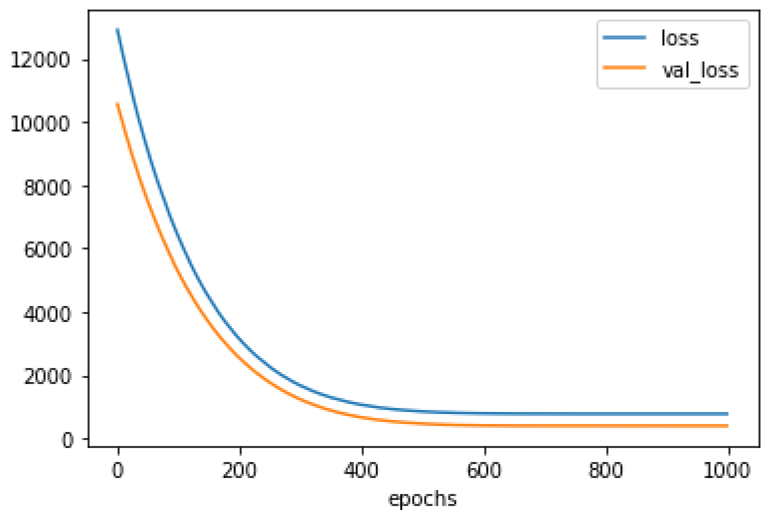
(continued from previous page)

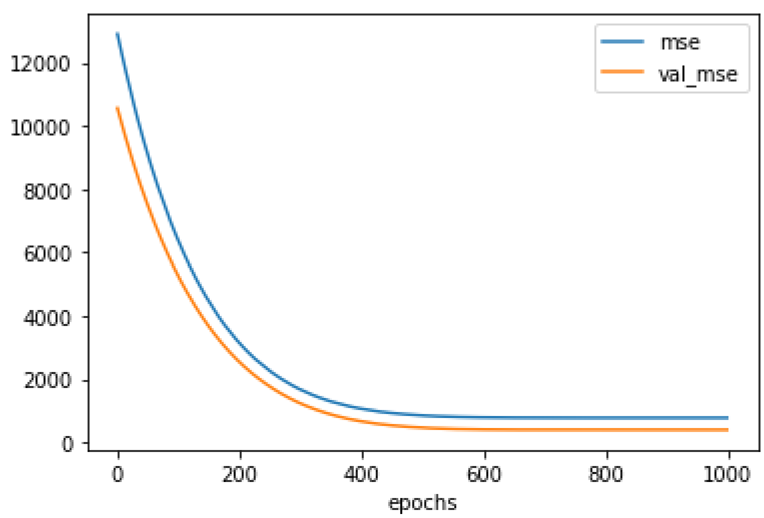
```
    batch_size=32,
    verbose=0,
    validation_split = 0.2)
```

[36]:
```
history_metrics = pd.DataFrame(history.history)
history_metrics['epochs'] = history.epoch
```

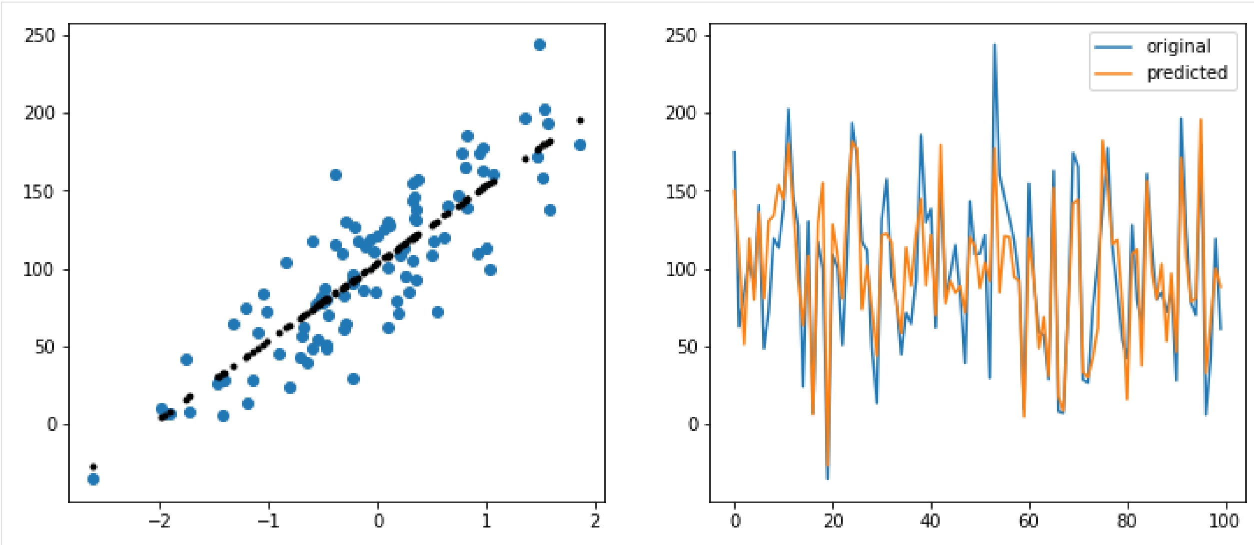
[38]:
```
history_metrics.plot(x='epochs',y=['loss','val_loss'])
history_metrics.plot(x='epochs',y=['mse','val_mse'])
```

[38]: <AxesSubplot:xlabel='epochs'>





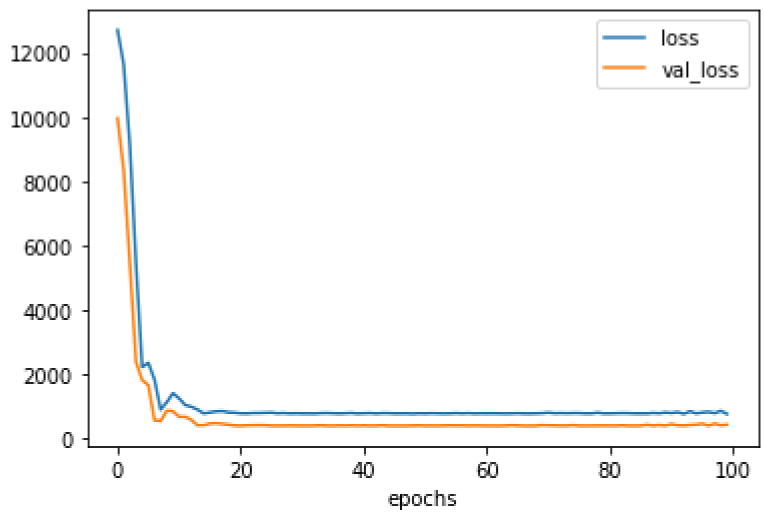
[39]:
```
y_hat = model.predict(df[['x0','x1']].values)

plot_regression(df.x1,df.y,y_hat)
```

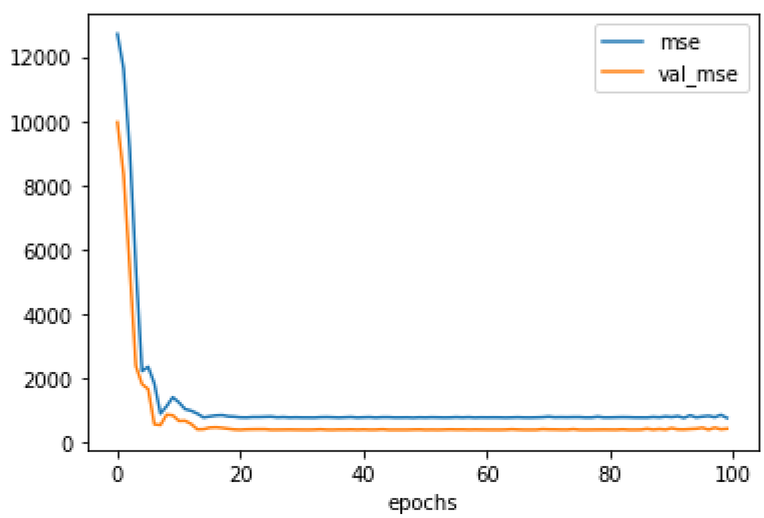


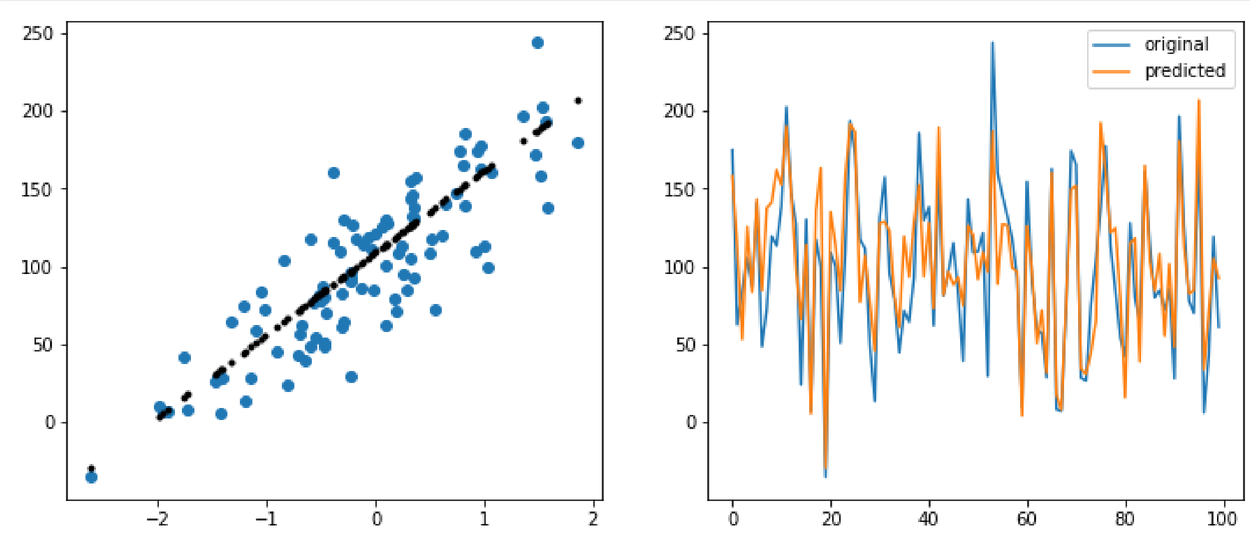
10.2.2 A little bit deep neural net but no activation functions

[40]:
```
model = tf.keras.Sequential([
    normalizer,
    tf.keras.layers.Dense(units=5),
    tf.keras.layers.Dense(units=5),
    tf.keras.layers.Dense(units=1)
])

model.summary()

model.compile(
    optimizer=tf.optimizers.Adam(learning_rate=0.1),
    loss=['mse'],
    metrics=['mse']
)

history = model.fit(
    df[['x0','x1']],
    df.y,
    epochs=100,
    batch_size=32,
    verbose=0,
    validation_split = 0.2)

history_metrics = pd.DataFrame(history.history)
history_metrics['epochs'] = history.epoch

history_metrics.plot(x='epochs',y=['loss','val_loss'])
history_metrics.plot(x='epochs',y=['mse','val_mse'])

y_hat = model.predict(df[['x0','x1']].values)

plot_regression(df.x1,df.y,y_hat)
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 normalization (Normalizatio  (None, 2)                5         
 n)                                                              
                                                                 
 dense_1 (Dense)             (None, 5)                 15        
                                                                 
 dense_2 (Dense)             (None, 5)                 30        
                                                                 
 dense_3 (Dense)             (None, 1)                 6         
                                                                 
=================================================================
Total params: 56
Trainable params: 51
Non-trainable params: 5
_________________________________________________________________
```



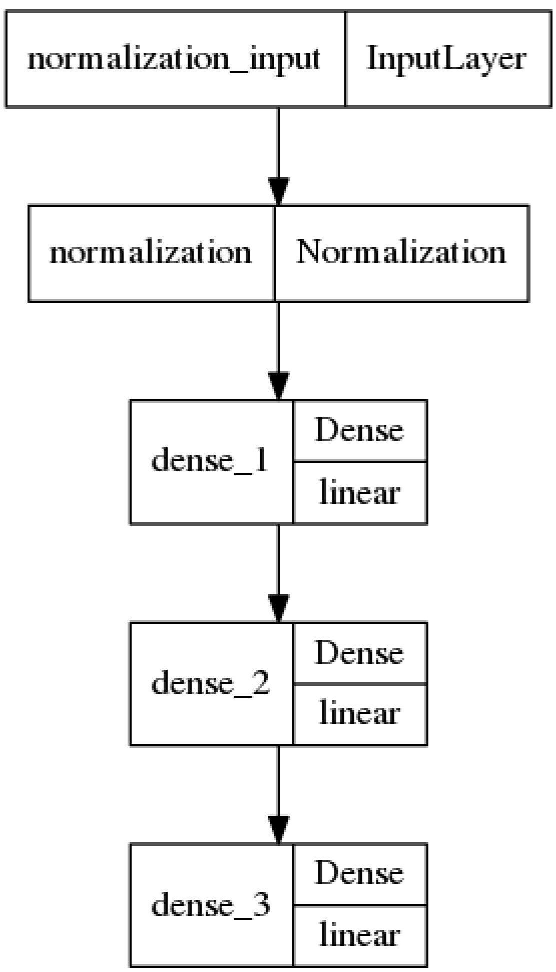




So I didn't introduce any activation/ non-linearity, and it is, no matter how deep the network is, a linear regression model. Ha Ha Ha

[41]:
```
tf.keras.utils.plot_model(model,show_layer_activations=True)
```

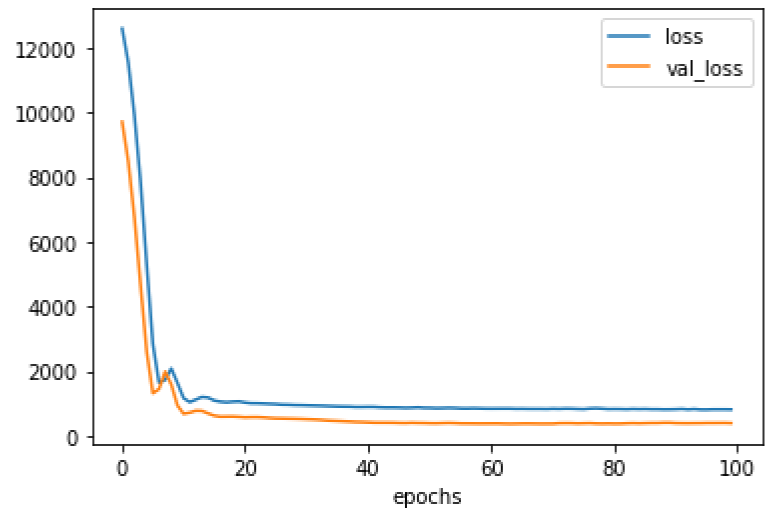
[41]:

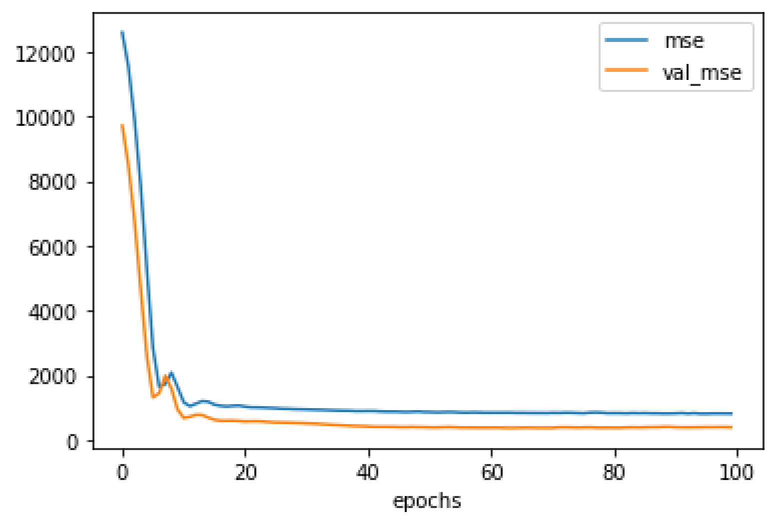


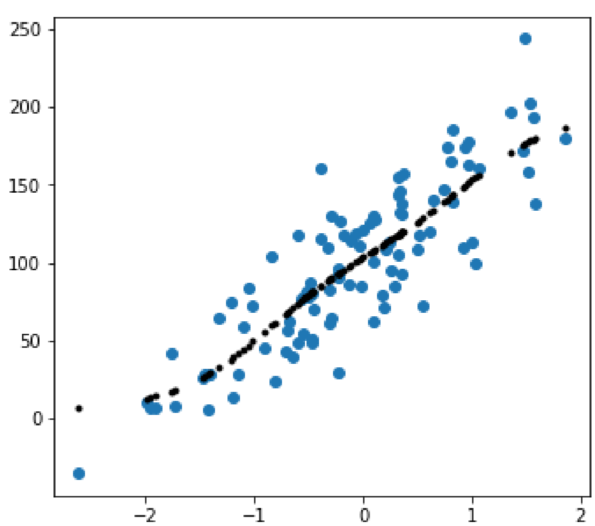
10.2.3 now a neural net with sigmoid applied

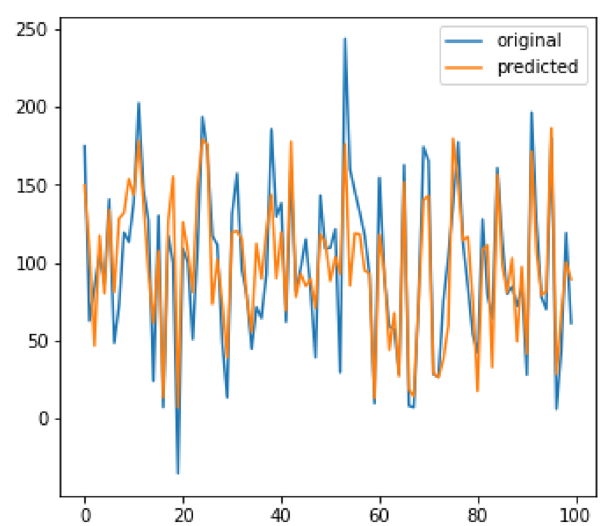
[42]:
```
model = tf.keras.Sequential([
    normalizer,
    tf.keras.layers.Dense(units=5,activation='sigmoid'),
    tf.keras.layers.Dense(units=5),
    tf.keras.layers.Dense(units=1)
])

model.summary()

model.compile(
    optimizer=tf.optimizers.Adam(learning_rate=0.1),
    loss=['mse'],
    metrics=['mse']
)

history = model.fit(
    df[['x0','x1']],
    df.y,
    epochs=100,
    batch_size=32,
    verbose=0,
    validation_split = 0.2)

history_metrics = pd.DataFrame(history.history)
history_metrics['epochs'] = history.epoch
history_metrics.plot(x='epochs',y=['loss','val_loss'])
history_metrics.plot(x='epochs',y=['mse','val_mse'])

y_hat = model.predict(df[['x0','x1']].values)

plot_regression(df.x1,df.y,y_hat)
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 normalization (Normalizatio  (None, 2)                5
 n)

 dense_4 (Dense)             (None, 5)                 15

 dense_5 (Dense)             (None, 5)                 30

 dense_6 (Dense)             (None, 1)                 6

=================================================================
Total params: 56
Trainable params: 51
Non-trainable params: 5
_________________________________________________________________
```





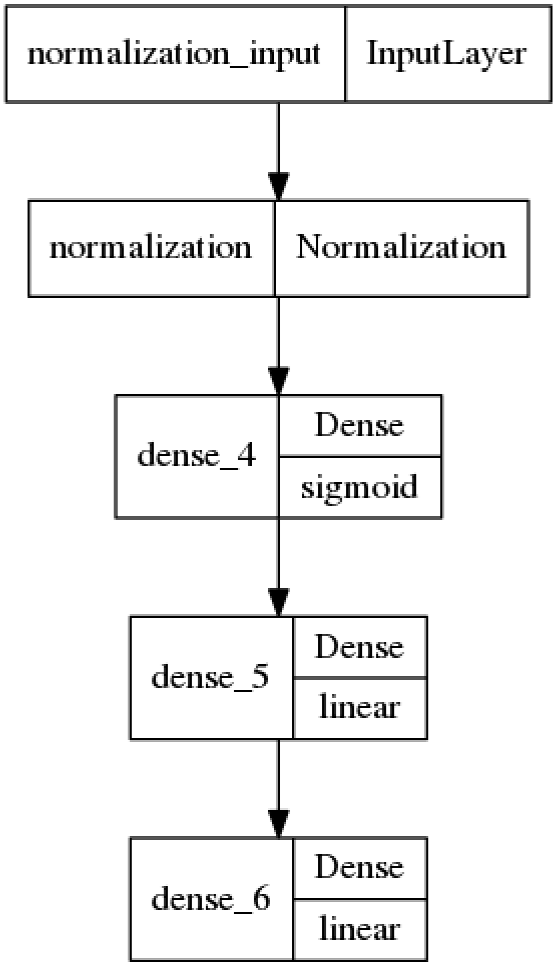




A little bit curved from sigmoid, trying to fit the pattern.

[43]:
```
tf.keras.utils.plot_model(model,show_layer_activations=True)
```

[43]:



10.2.4 2 sigmoids applied in the net

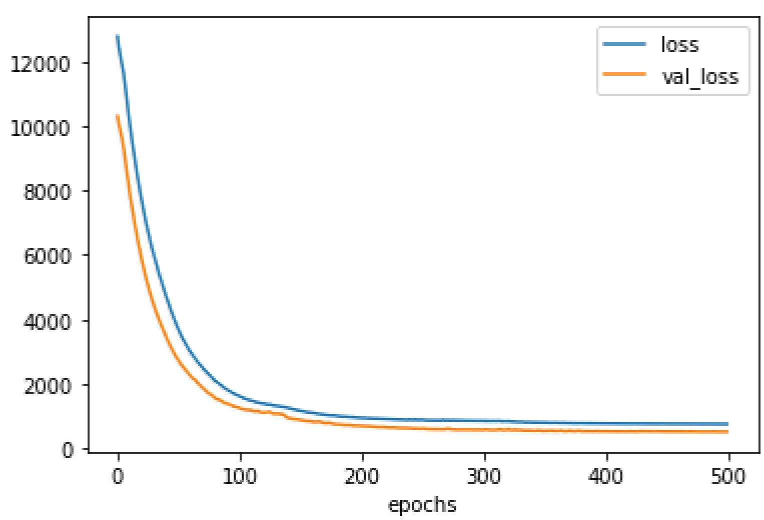
[44]:
```
model = tf.keras.Sequential([
    normalizer,
    tf.keras.layers.Dense(units=5,activation='sigmoid'),
    tf.keras.layers.Dense(units=5,activation='sigmoid'),
    tf.keras.layers.Dense(units=1)
])

model.summary()

model.compile(
    optimizer=tf.optimizers.Adam(learning_rate=0.1),
    loss=['mse'],
    metrics=['mse']
)

history = model.fit(
```

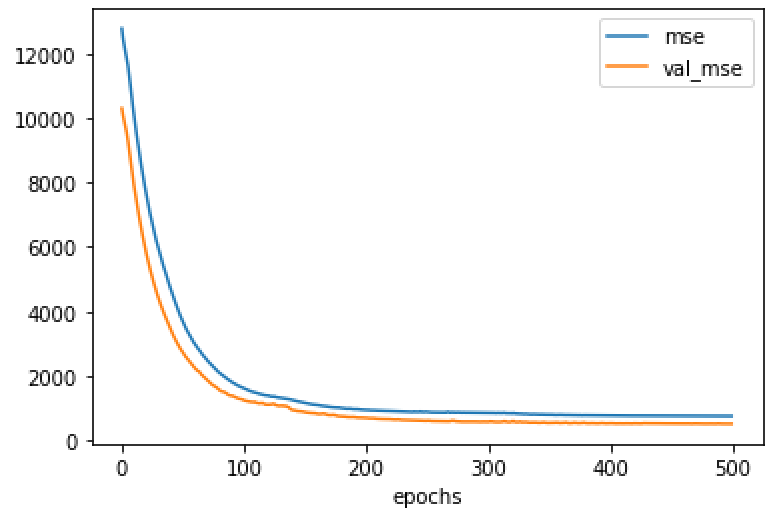
(continues on next page)

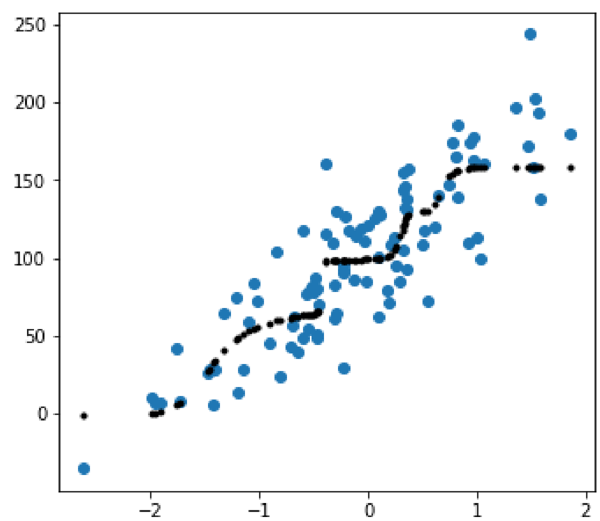
(continued from previous page)

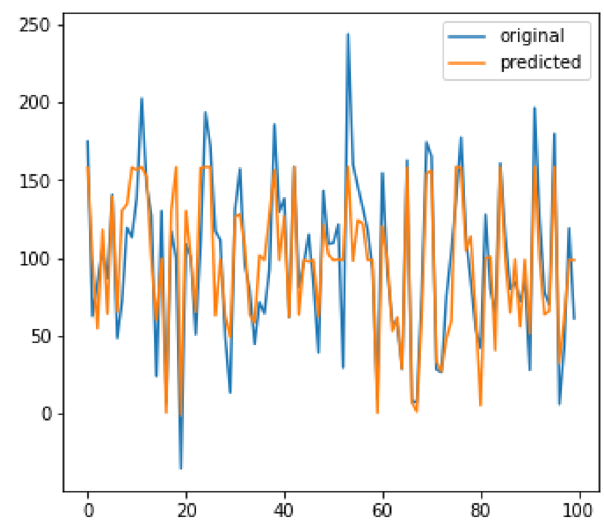
```
    df[['x0','x1']],
    df.y,
    epochs=500,
    batch_size=32,
    verbose=0,
    validation_split = 0.2)

history_metrics = pd.DataFrame(history.history)
history_metrics['epochs'] = history.epoch
history_metrics.plot(x='epochs',y=['loss','val_loss'])
history_metrics.plot(x='epochs',y=['mse','val_mse'])

y_hat = model.predict(df[['x0','x1']].values)

plot_regression(df.x1,df.y,y_hat)
```

```
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 normalization (Normalizatio  (None, 2)                5         
 n)                                                              
                                                                 
 dense_7 (Dense)             (None, 5)                 15        
                                                                 
 dense_8 (Dense)             (None, 5)                 30        
                                                                 
 dense_9 (Dense)             (None, 1)                 6         
                                                                 
=================================================================
Total params: 56
Trainable params: 51
Non-trainable params: 5
_________________________________________________________________
```





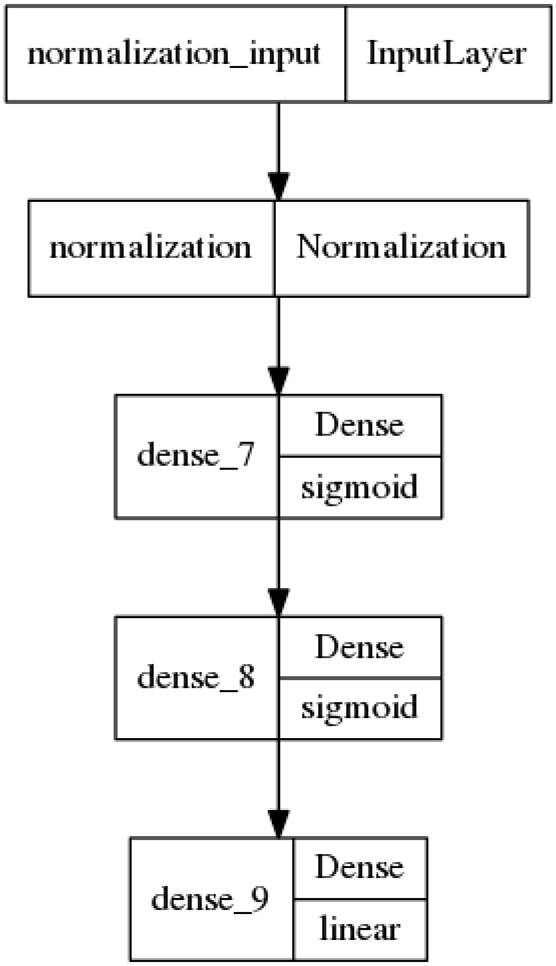




more curves/ non-linear pattern matching, with increasing sigmoid layers.

[45]:
```
tf.keras.utils.plot_model(model,show_layer_activations=True)
```

[45]:



10.3 lets try with a little bit complex pattern

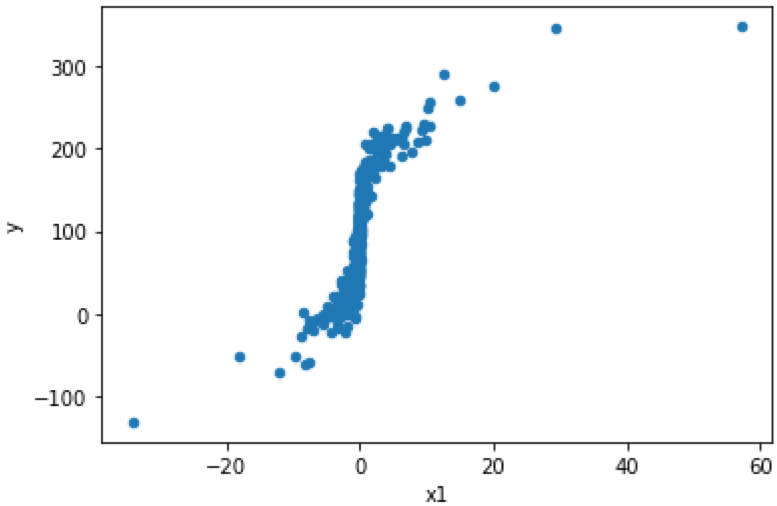
[48]:
```
X, y = make_regression(n_features=1,noise=20,random_state=42,bias=100,n_samples=500)

df = pd.DataFrame()
df['x1'] = X[...,-1]**3
df['y'] = y
df['x0'] = 1
df.head()
```

[48]:

	x1	y	x0
0	-0.528099	57.401862	1
1	0.000913	102.950676	1
2	0.105983	123.553604	1
3	-3.232089	-9.967066	1
4	-0.057206	77.788884	1

[49]:
```
df.plot(x='x1',y='y',kind='scatter')
```

[49]:
```
<AxesSubplot:xlabel='x1', ylabel='y'>
```



10.3.1 a completely linear model for complex data

[50]:
```
normalizer = tf.keras.layers.Normalization(axis=-1)
normalizer.adapt(df[['x0','x1']].values) # adapt is like fit
```

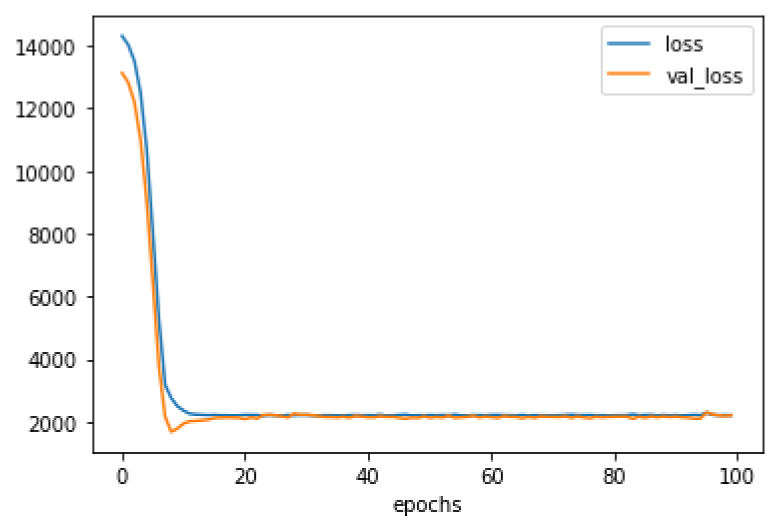
[51]:
```
model = tf.keras.Sequential([
    normalizer,
    tf.keras.layers.Dense(units=5),
    tf.keras.layers.Dense(units=5),
    tf.keras.layers.Dense(units=1)
])

model.summary()

model.compile(
    optimizer=tf.optimizers.Adam(learning_rate=0.01),
    loss=['mse'],
    metrics=['mse']
)

history = model.fit(
    df[['x0','x1']],
    df.y,
    epochs=100,
    batch_size=32,
    verbose=0,
    validation_split = 0.2
)

history_metrics = pd.DataFrame(history.history)
history_metrics['epochs'] = history.epoch
history_metrics.plot(x='epochs',y=['loss','val_loss'])
```

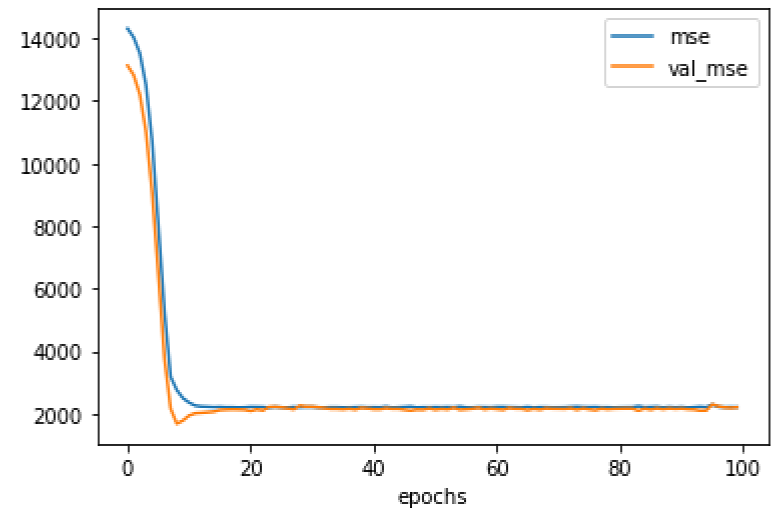
(continues on next page)

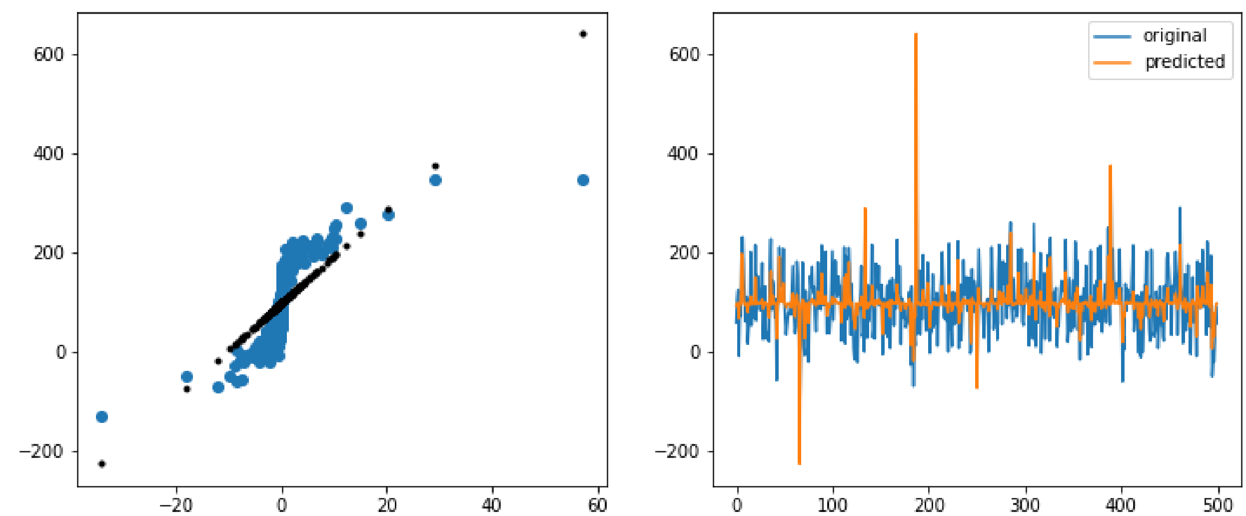
(continued from previous page)

```
history_metrics.plot(x='epochs',y=['mse','val_mse'])

y_hat = model.predict(df[['x0','x1']].values)

plot_regression(df.x1,df.y,y_hat)
```

```
Model: "sequential_4"
_________________________________________________________________
 Layer (type)                Output Shape              Param #   
=================================================================
 normalization_1 (Normalizat  (None, 2)                5         
 ion)                                                            
                                                                 
 dense_10 (Dense)            (None, 5)                 15        
                                                                 
 dense_11 (Dense)            (None, 5)                 30        
                                                                 
 dense_12 (Dense)            (None, 1)                 6         
                                                                 
=================================================================
Total params: 56
Trainable params: 51
Non-trainable params: 5
_________________________________________________________________
```



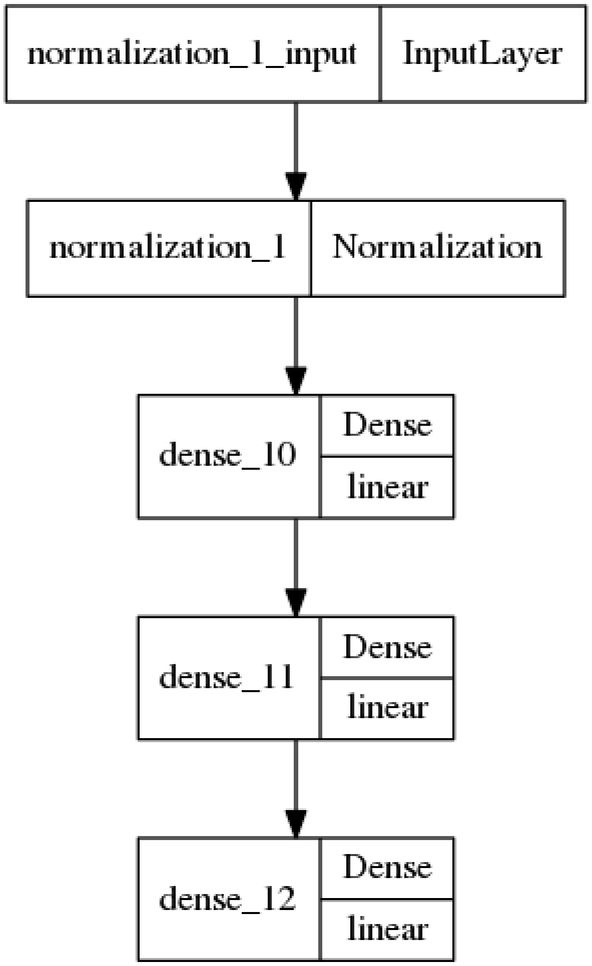




As expected, no matter how deep it is, it matches a linear pattern.

[52]:
```
tf.keras.utils.plot_model(model,show_layer_activations=True)
```

[52]:



10.3.2 with a sigmoid introducing non linearity

[53]:

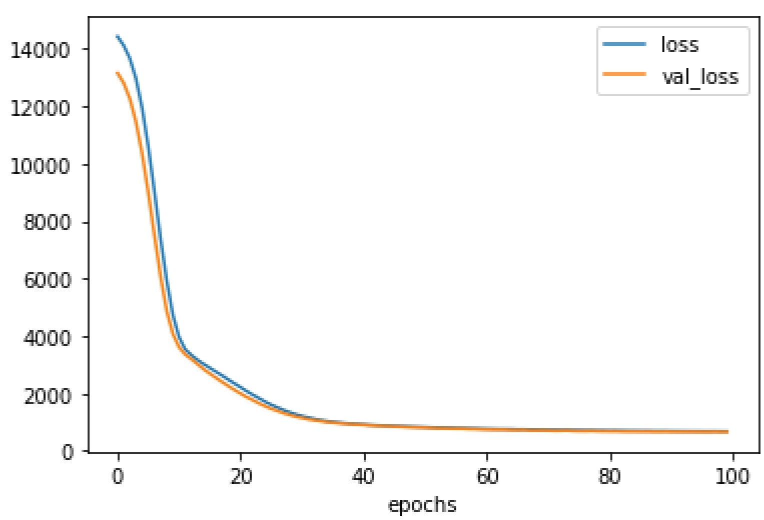
```
model = tf.keras.Sequential([
    normalizer,
    tf.keras.layers.Dense(units=5, activation='sigmoid'),
    tf.keras.layers.Dense(units=5),
    tf.keras.layers.Dense(units=1)
])

model.summary()

model.compile(
    optimizer=tf.optimizers.Adam(learning_rate=0.01),
    loss=['mse'],
    metrics=['mse']
)

history = model.fit(
    df[['x0','x1']],
```

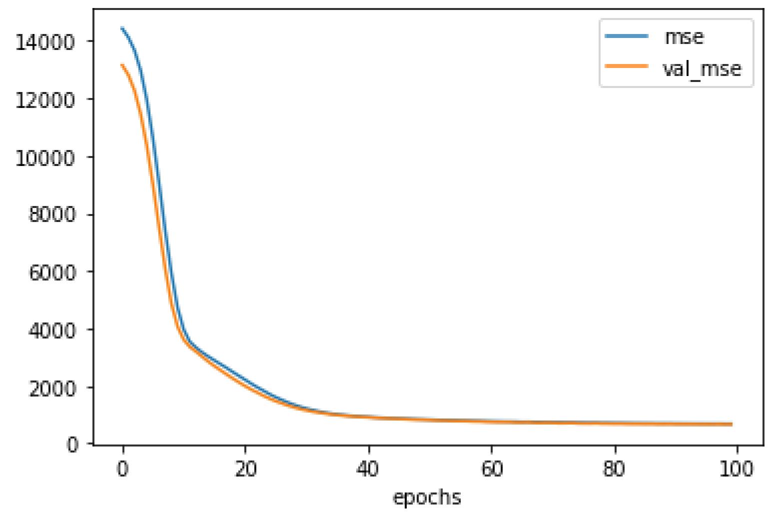
(continues on next page)

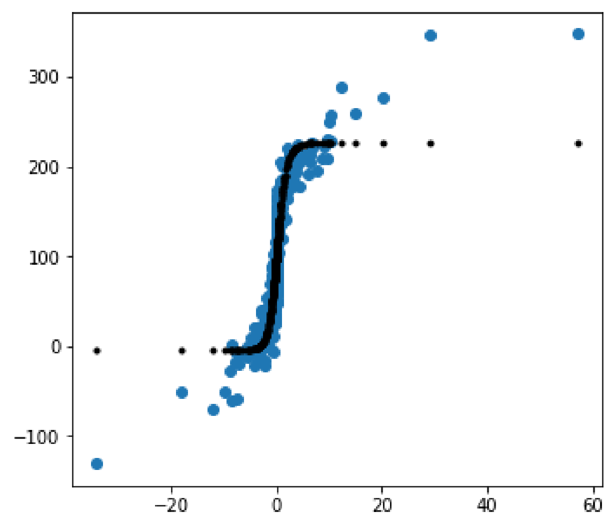
(continued from previous page)

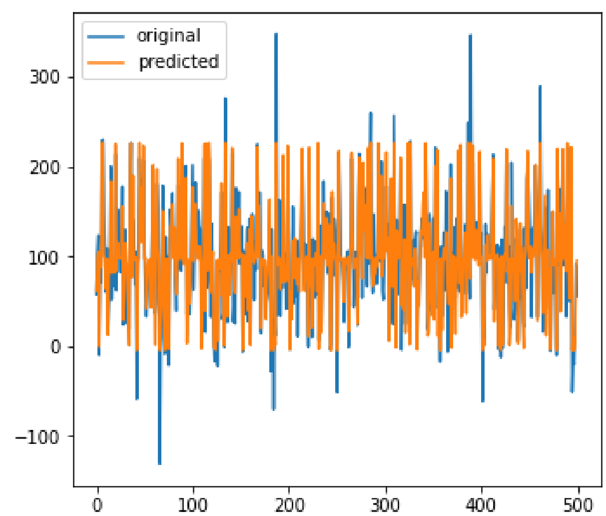
```
    df.y,
    epochs=100,
    batch_size=32,
    verbose=0,
    validation_split = 0.2
)

history_metrics = pd.DataFrame(history.history)
history_metrics['epochs'] = history.epoch
history_metrics.plot(x='epochs',y=['loss','val_loss'])
history_metrics.plot(x='epochs',y=['mse','val_mse'])

y_hat = model.predict(df[['x0','x1']].values)

plot_regression(df.x1,df.y,y_hat)
```

```
Model: "sequential_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 normalization_1 (Normalizat  (None, 2)                5
 ion)

 dense_13 (Dense)            (None, 5)                 15

 dense_14 (Dense)            (None, 5)                 30

 dense_15 (Dense)            (None, 1)                 6

=================================================================
Total params: 56
Trainable params: 51
Non-trainable params: 5
_________________________________________________________________
```



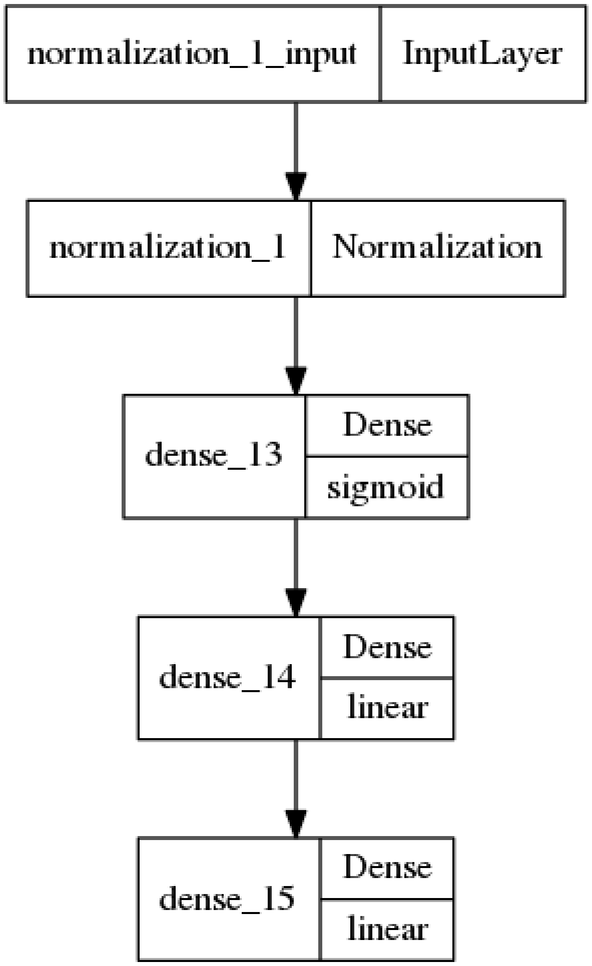






[54]:
```
tf.keras.utils.plot_model(model,show_layer_activations=True)
```

[54]:



10.3.3 with a relu layer

[57]:

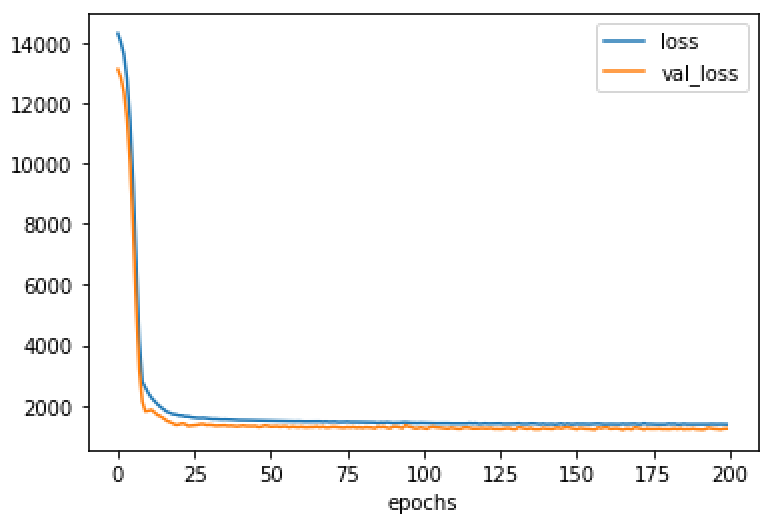
```
model = tf.keras.Sequential([
    normalizer,
    tf.keras.layers.Dense(units=5, activation='relu'),
    tf.keras.layers.Dense(units=5),
    tf.keras.layers.Dense(units=1)
])

model.summary()

model.compile(
    optimizer=tf.optimizers.Adam(learning_rate=0.01),
    loss=['mse'],
    metrics=['mse']
)

history = model.fit(
    df[['x0','x1']],
```

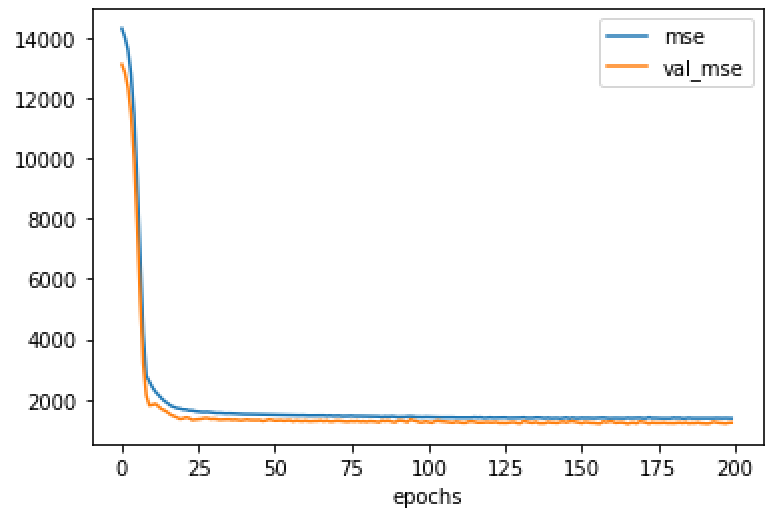
(continues on next page)

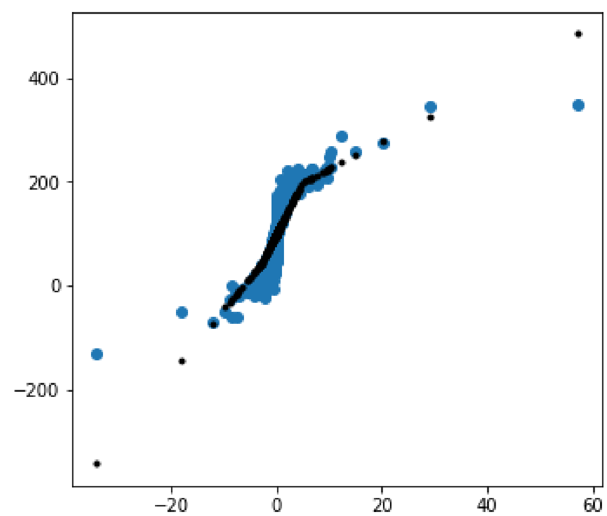
(continued from previous page)

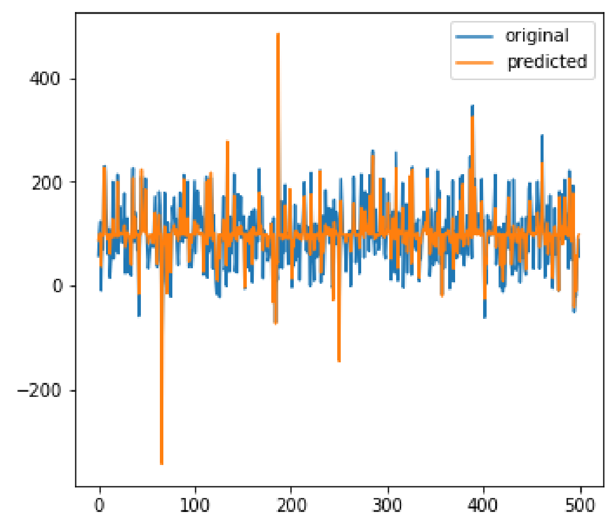
```
    df.y,
    epochs=200,
    batch_size=32,
    verbose=0,
    validation_split = 0.2
)

history_metrics = pd.DataFrame(history.history)
history_metrics['epochs'] = history.epoch
history_metrics.plot(x='epochs',y=['loss','val_loss'])
history_metrics.plot(x='epochs',y=['mse','val_mse'])

y_hat = model.predict(df[['x0','x1']].values)

plot_regression(df.x1,df.y,y_hat)
```

```
Model: "sequential_8"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 normalization_1 (Normalizat  (None, 2)                5
 ion)

 dense_22 (Dense)            (None, 5)                 15

 dense_23 (Dense)            (None, 5)                 30

 dense_24 (Dense)            (None, 1)                 6

=================================================================
Total params: 56
Trainable params: 51
Non-trainable params: 5
_________________________________________________________________
```



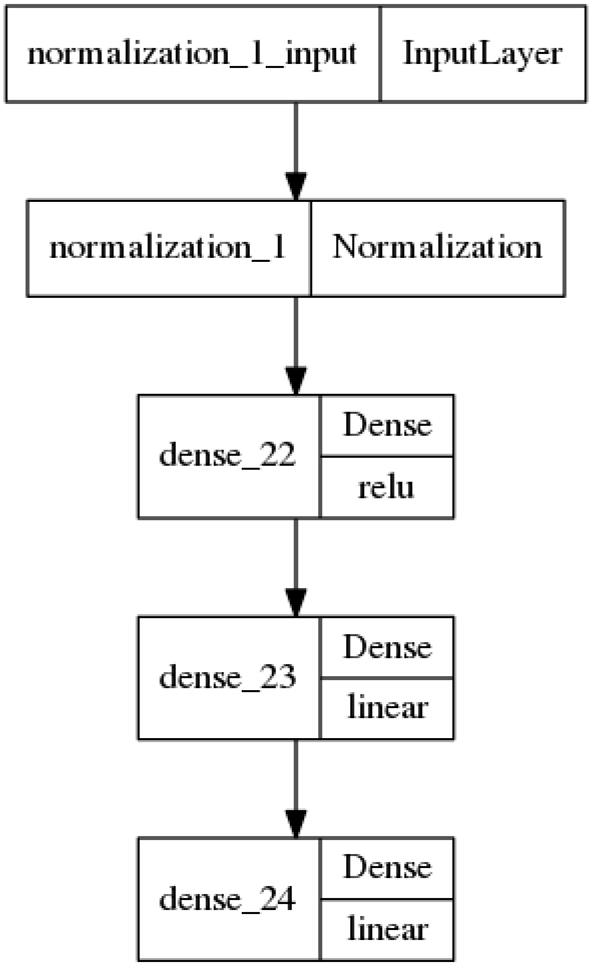






[58]:
```
tf.keras.utils.plot_model(model,show_layer_activations=True)
```

[58]:



10.3.4 with two relu layers

[59]:

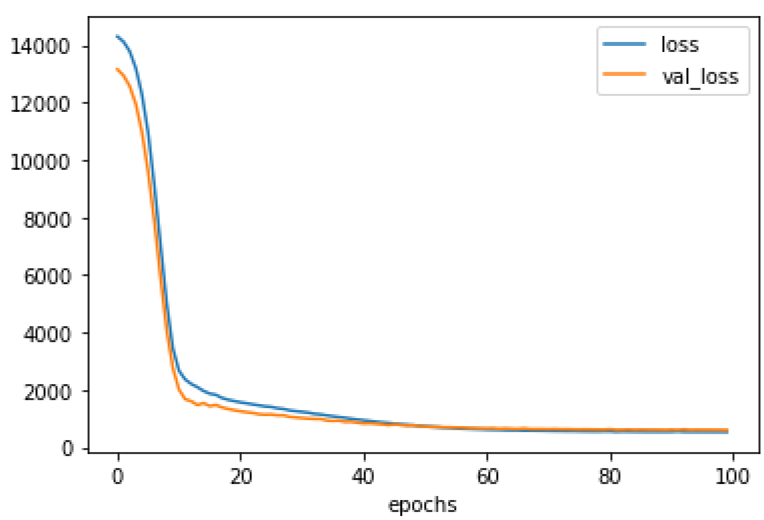
```
model = tf.keras.Sequential([
    normalizer,
    tf.keras.layers.Dense(units=5, activation='relu'),
    tf.keras.layers.Dense(units=5, activation='relu'),
    tf.keras.layers.Dense(units=1)
])

model.summary()

model.compile(
    optimizer=tf.optimizers.Adam(learning_rate=0.01),
    loss=['mse'],
    metrics=['mse']
)

history = model.fit(
    df[['x0','x1']],
```

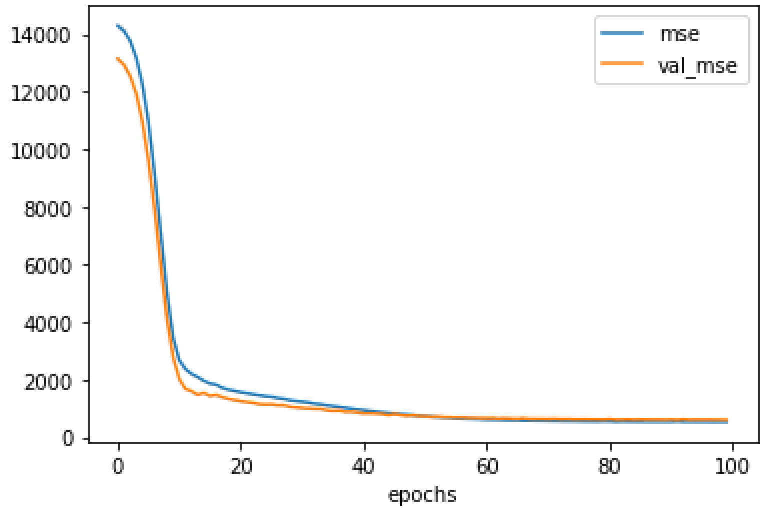
(continues on next page)

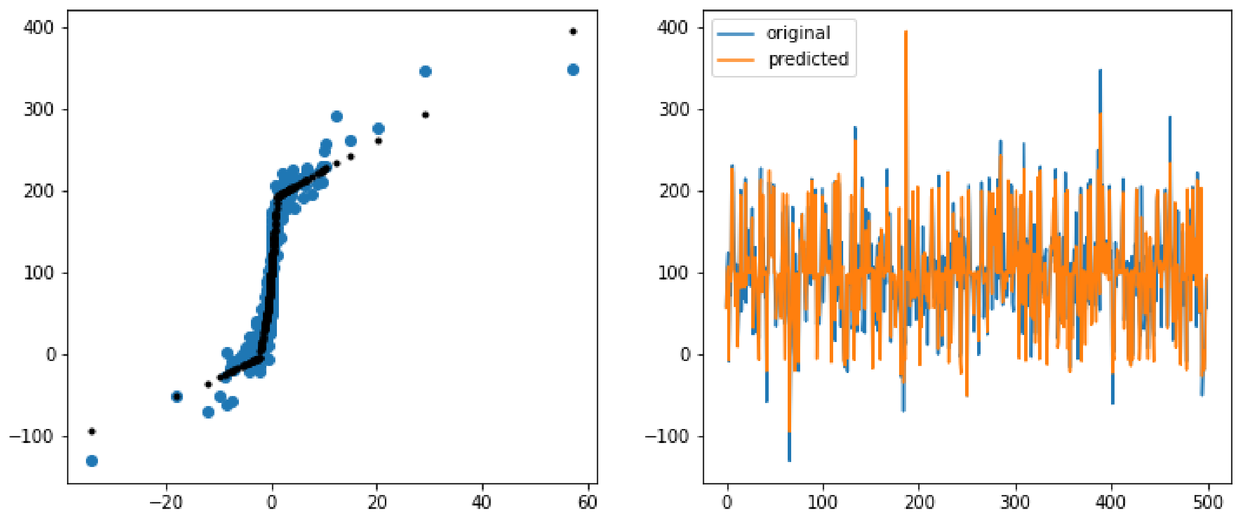
(continued from previous page)

```
    df.y,
    epochs=100,
    batch_size=32,
    verbose=0,
    validation_split = 0.2
)

history_metrics = pd.DataFrame(history.history)
history_metrics['epochs'] = history.epoch
history_metrics.plot(x='epochs',y=['loss','val_loss'])
history_metrics.plot(x='epochs',y=['mse','val_mse'])

y_hat = model.predict(df[['x0','x1']].values)

plot_regression(df.x1,df.y,y_hat)
```

```
Model: "sequential_9"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 normalization_1 (Normalizat  (None, 2)                5
 ion)

 dense_25 (Dense)            (None, 5)                 15

 dense_26 (Dense)            (None, 5)                 30

 dense_27 (Dense)            (None, 1)                 6

=================================================================
Total params: 56
Trainable params: 51
Non-trainable params: 5
_________________________________________________________________
```

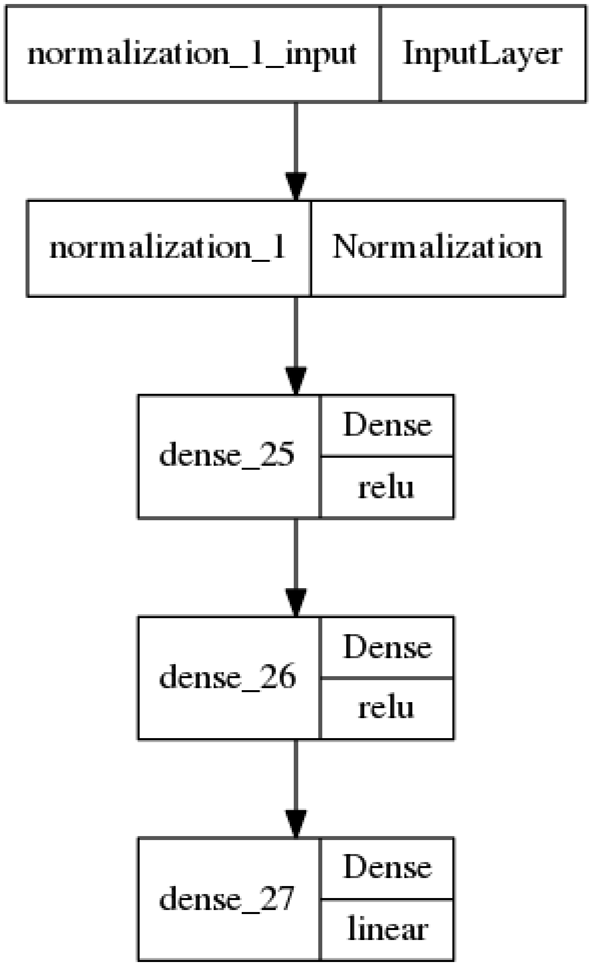






[60]:
```
tf.keras.utils.plot_model(model,show_layer_activations=True)
```

[60]:



10.4 sine wave with a nerual network

[61]:

```
X, y = make_regression(n_features=1,noise=10,random_state=42,n_samples=500)

X.shape, y.shape
```

[61]:

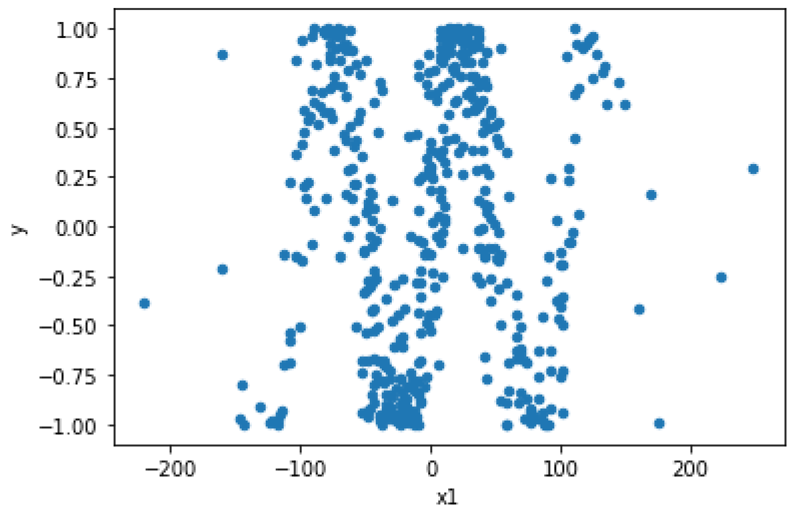
```
((500, 1), (500,))
```

[62]:

```
df = pd.DataFrame()
df['x1'] = y
df['y'] = np.sin(X[...,-1]*4)
df['x0'] = 1
df.plot(x='x1',y='y',kind='scatter')
```

[62]:

```
<AxesSubplot:xlabel='x1', ylabel='y'>
```



10.4.1 Trying out with linear model

I know this is not gonna work.

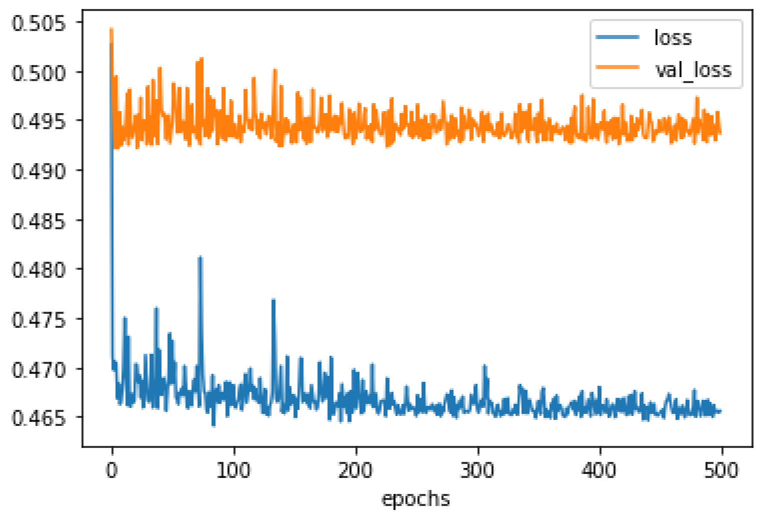
[63]:
```
normalizer = tf.keras.layers.Normalization(axis=-1)
normalizer.adapt(df[['x0','x1']].values) # adapt is like fit

model = tf.keras.Sequential([
    normalizer,
    tf.keras.layers.Dense(units=5, activation='linear'),
    tf.keras.layers.Dense(units=1)
])

model.summary()

model.compile(
    optimizer=tf.optimizers.Adam(learning_rate=0.01),
    loss=['mse'],
    metrics=['mse']
)

history = model.fit(df[['x0','x1']], df.y, epochs=500, \
                    batch_size=32, verbose=0, validation_split = 0.2)

history_metrics = pd.DataFrame(history.history)
history_metrics['epochs'] = history.epoch
history_metrics.plot(x='epochs',y=['loss','val_loss'])
history_metrics.plot(x='epochs',y=['mse','val_mse'])

y_hat = model.predict(df[['x0','x1']].values)

plot_regression(df.x1,df.y,y_hat)
```

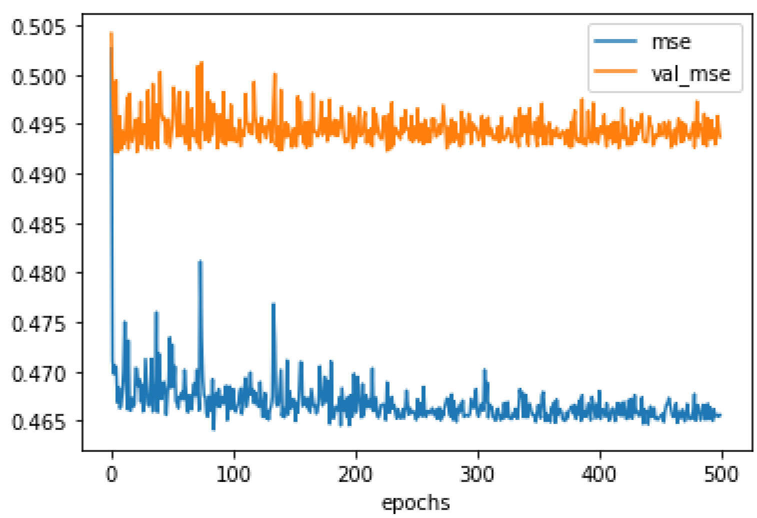
```
Model: "sequential_10"
_________________________________________________________________
```

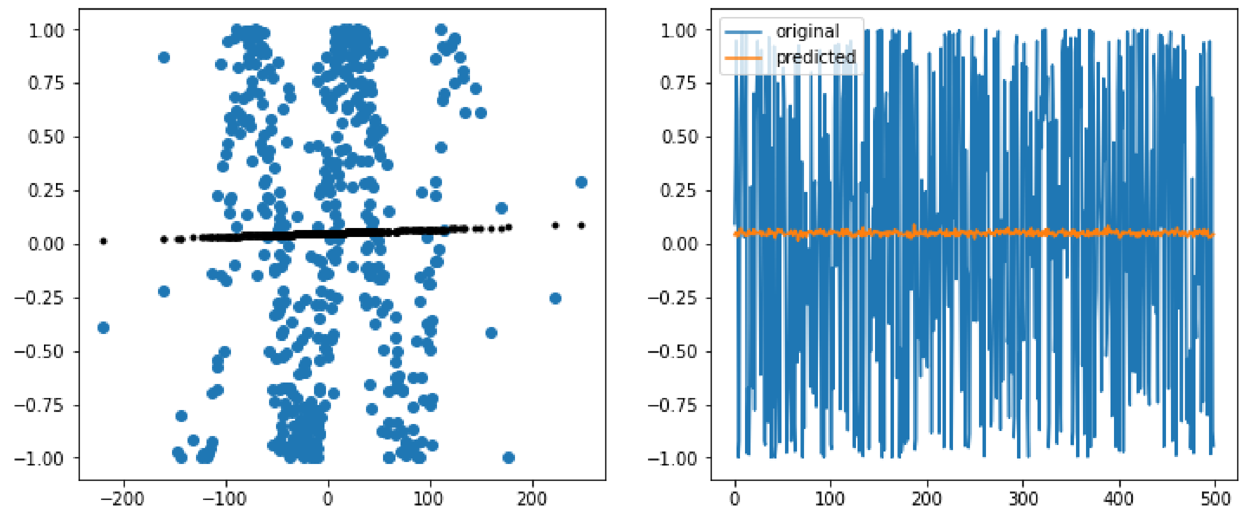
(continues on next page)

(continued from previous page)

```
 Layer (type)                Output Shape              Param #
=================================================================
 normalization_2 (Normalizat  (None, 2)                5
 ion)

 dense_28 (Dense)            (None, 5)                 15

 dense_29 (Dense)            (None, 1)                 6

=================================================================
Total params: 26
Trainable params: 21
Non-trainable params: 5
_________________________________________________________________
```



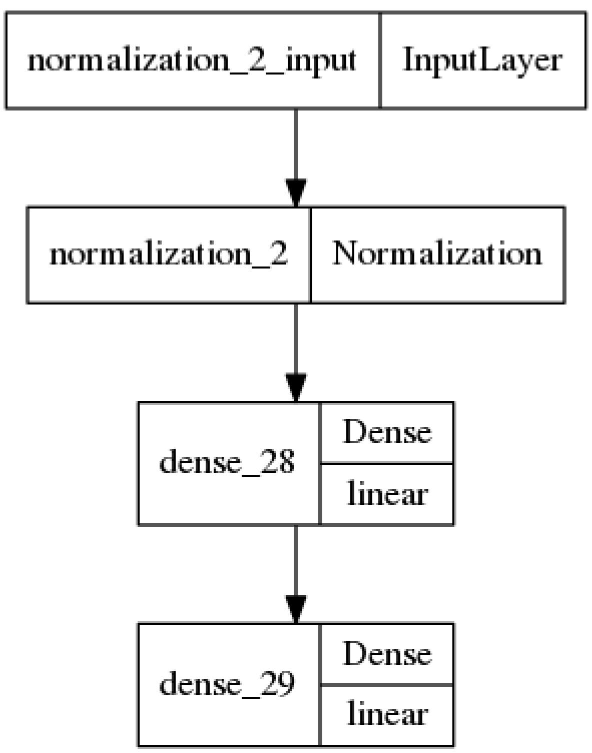




Pretty obvious.

[64]:
```
tf.keras.utils.plot_model(model,show_layer_activations=True)
```

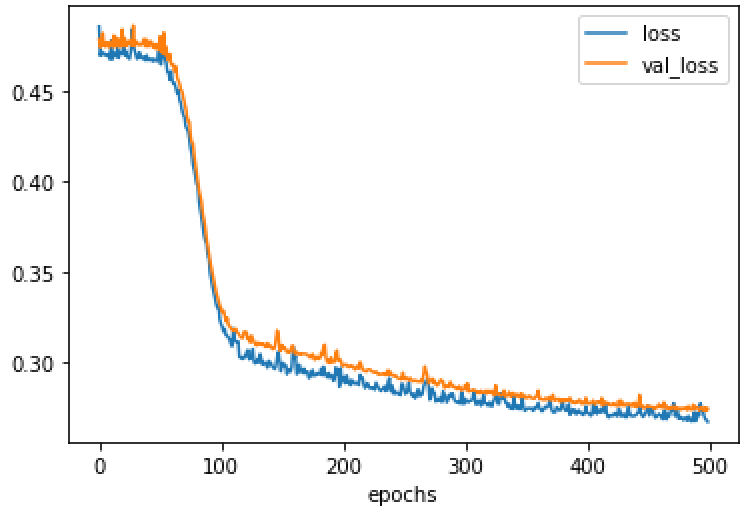
[64]:

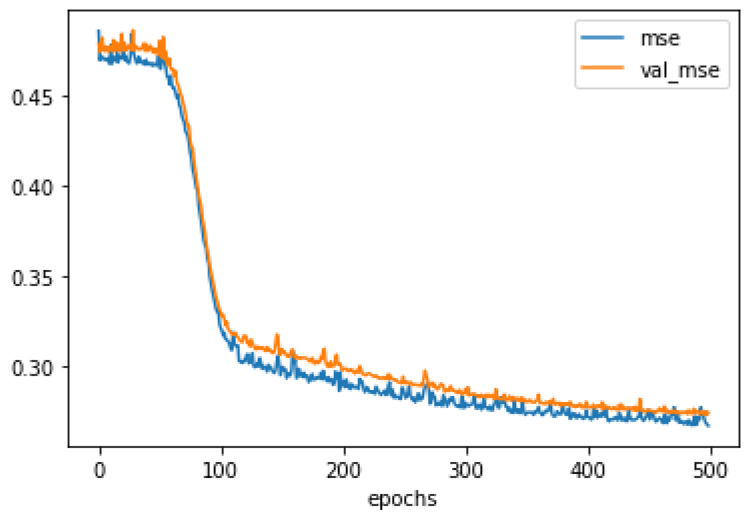


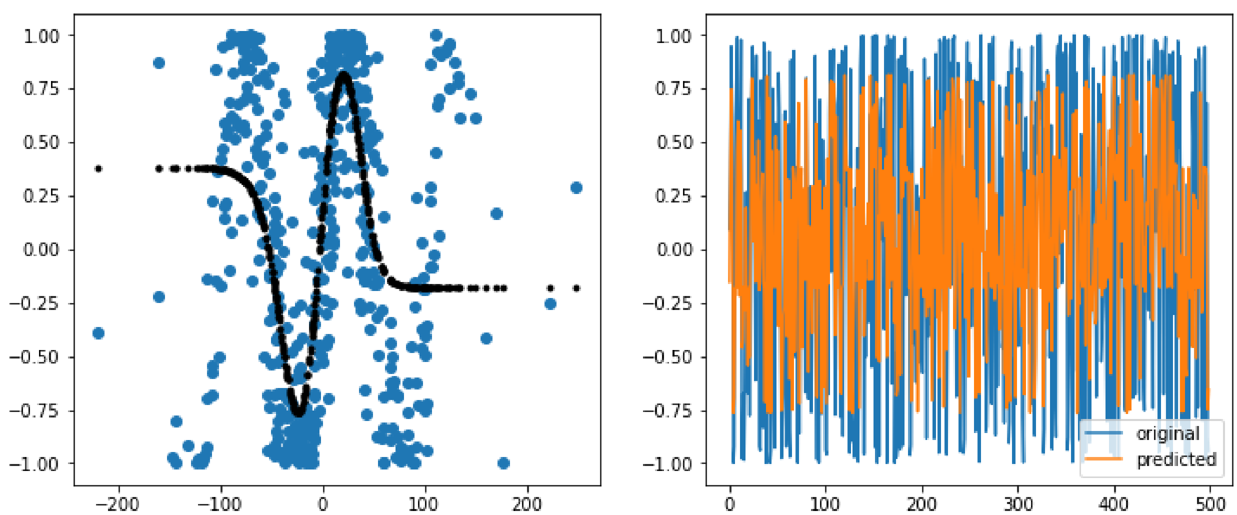
10.4.2 Now with 2 sigmoid layers

[65]:
```
normalizer = tf.keras.layers.Normalization(axis=-1)
normalizer.adapt(df[['x0','x1']].values) # adapt is like fit

model = tf.keras.Sequential([
    normalizer,
    tf.keras.layers.Dense(units=5, activation='sigmoid'),
    tf.keras.layers.Dense(units=5, activation='sigmoid'),
    tf.keras.layers.Dense(units=1)
])

model.summary()

model.compile(
    optimizer=tf.optimizers.Adam(learning_rate=0.01),
    loss=['mse'],
    metrics=['mse']
)

history = model.fit(df[['x0','x1']], df.y, epochs=500, \
                    batch_size=32, verbose=0, validation_split = 0.3)

history_metrics = pd.DataFrame(history.history)
history_metrics['epochs'] = history.epoch
history_metrics.plot(x='epochs',y=['loss','val_loss'])
history_metrics.plot(x='epochs',y=['mse','val_mse'])

y_hat = model.predict(df[['x0','x1']].values)

plot_regression(df.x1,df.y,y_hat)
```

```
Model: "sequential_11"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 normalization_3 (Normalizat  (None, 2)                5
 ion)

 dense_30 (Dense)            (None, 5)                 15

 dense_31 (Dense)            (None, 5)                 30

 dense_32 (Dense)            (None, 1)                 6

=================================================================
Total params: 56
Trainable params: 51
Non-trainable params: 5
_________________________________________________________________
```

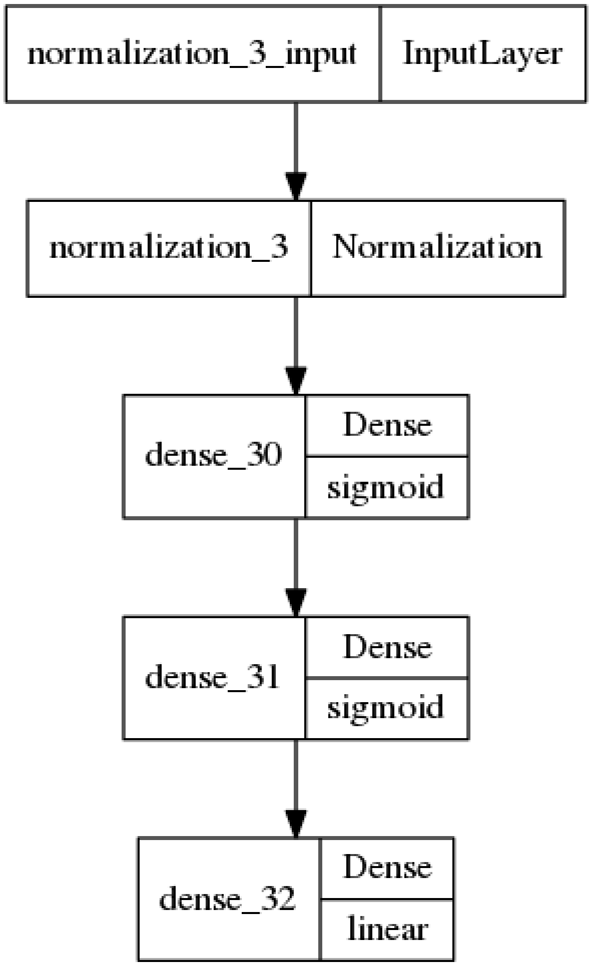






[66]:
```
tf.keras.utils.plot_model(model,show_layer_activations=True)
```

[66]:



10.4.3 6 layers neural network with sigmoid and tanh

[69]:

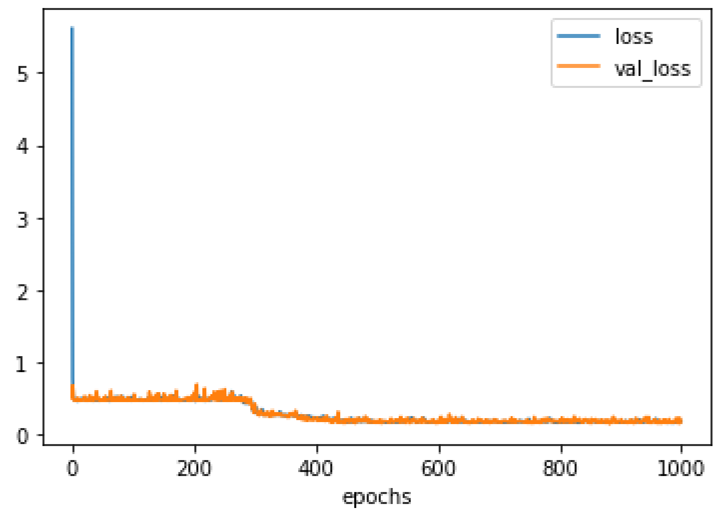
```
normalizer = tf.keras.layers.Normalization(axis=-1)
normalizer.adapt(df[['x0','x1']].values) # adapt is like fit

model = tf.keras.Sequential([
    normalizer,
    tf.keras.layers.Dense(units=100, activation='sigmoid'),
    tf.keras.layers.Dense(units=100, activation='sigmoid'),
    tf.keras.layers.Dense(units=100, activation='sigmoid'),
    tf.keras.layers.Dense(units=100, activation='tanh'),
    tf.keras.layers.Dense(units=100, activation='tanh'),
    tf.keras.layers.Dense(units=1)
])

model.summary()

model.compile(
    optimizer=tf.optimizers.Adam(learning_rate=0.01),
```

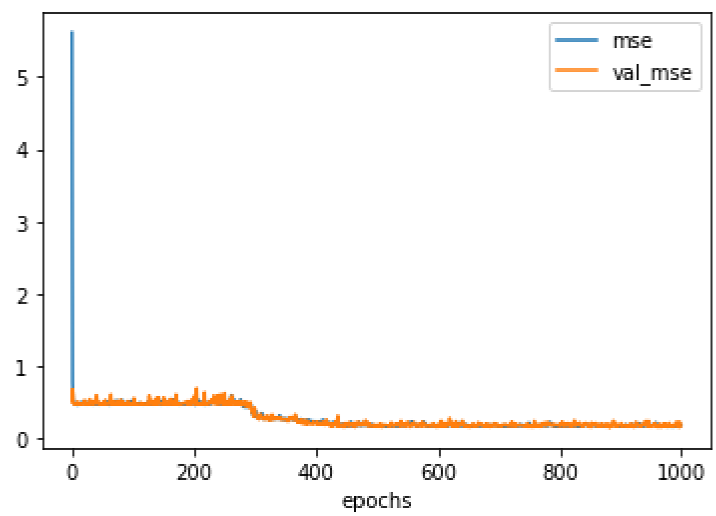
(continues on next page)

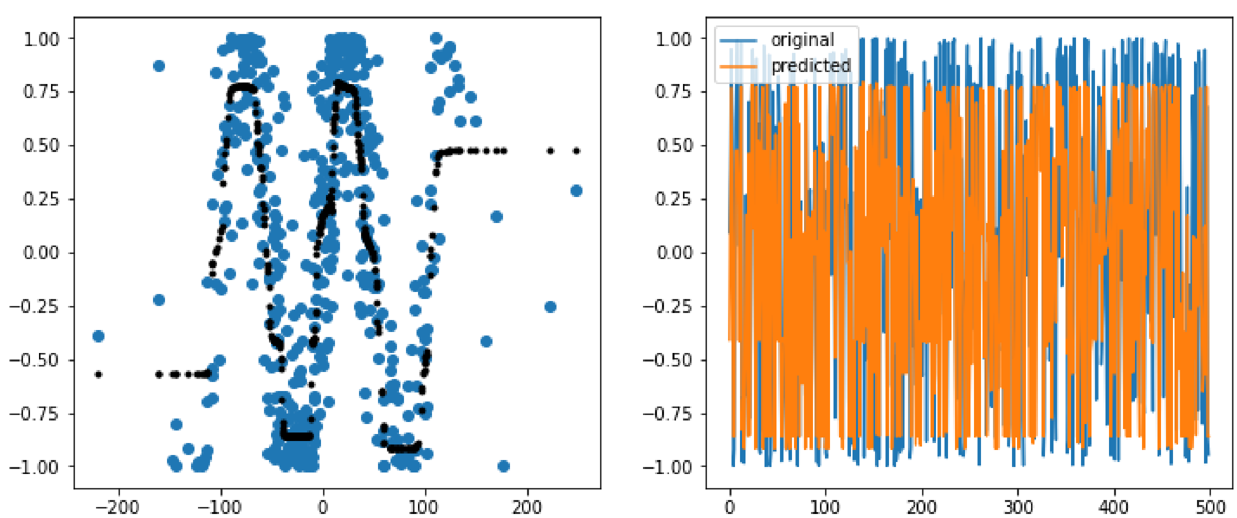
(continued from previous page)

```
    loss=['mse'],
    metrics=['mse']
)

history = model.fit(df[['x0','x1']], df.y, epochs=1000, \
                    batch_size=32, verbose=0, validation_split = 0.3)

history_metrics = pd.DataFrame(history.history)
history_metrics['epochs'] = history.epoch
history_metrics.plot(x='epochs',y=['loss','val_loss'])
history_metrics.plot(x='epochs',y=['mse','val_mse'])

y_hat = model.predict(df[['x0','x1']].values)

plot_regression(df.x1,df.y,y_hat)
```

```
Model: "sequential_13"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 normalization_5 (Normalizat  (None, 2)                5
 ion)

 dense_39 (Dense)            (None, 100)               300

 dense_40 (Dense)            (None, 100)               10100

 dense_41 (Dense)            (None, 100)               10100

 dense_42 (Dense)            (None, 100)               10100

 dense_43 (Dense)            (None, 100)               10100

 dense_44 (Dense)            (None, 1)                 101

=================================================================
Total params: 40,806
Trainable params: 40,801
Non-trainable params: 5
_________________________________________________________________
```

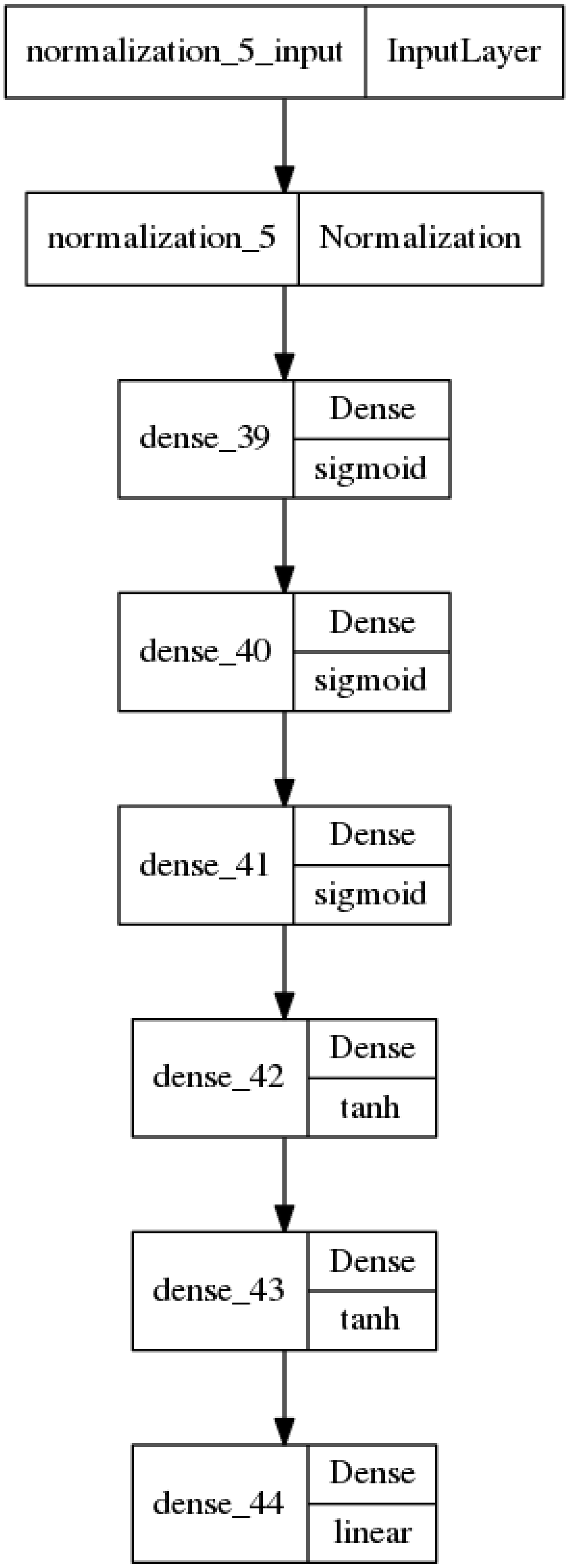






[71]:
```
tf.keras.utils.plot_model(model,show_layer_activations=True)
```

[71]:


normalization_5_input
InputLayer
normalization_5
Normalization
dense_39
Dense
sigmoid
dense_40
Dense
sigmoid
dense_41
Dense
sigmoid
dense_42
Dense
tanh
dense_43
Dense
tanh
dense_44
Dense
linear

CHAPTER

ELEVEN

CLASSIFICATION BOUNDARIES

[2]:
```
import numpy as np
import pandas as pd

import matplotlib.pyplot as plt
import seaborn as sns
from mpl_toolkits import mplot3d
from graphpkg.static import plot_classification_boundary
import tensorflow as tf
from sklearn.datasets import make_blobs, make_classification, make_regression
import warnings

warnings.filterwarnings("ignore")
```

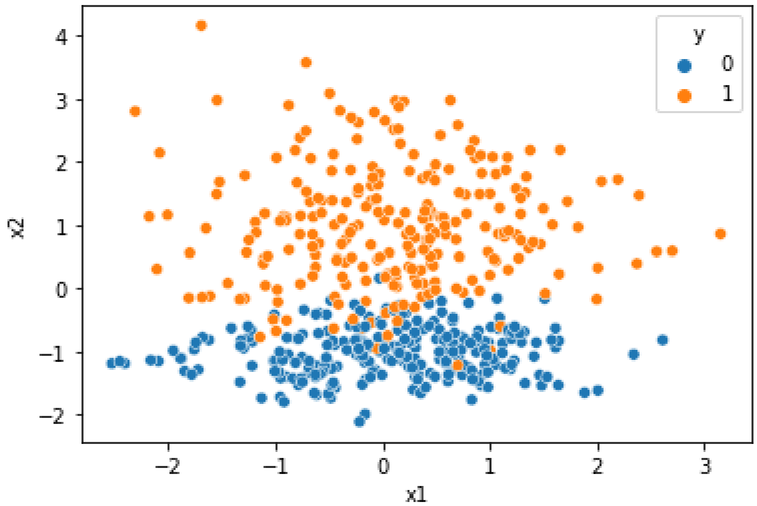
11.1 Basic Classification

Target here to generate a data that has only two classes and it can be classified by a linear model also. Like a linear hyperplane can also be a good decision boundary.

[2]:
```
X, y = make_classification(n_samples=500, n_features=2, random_state=30, \
                           n_informative=1, n_classes=2, n_clusters_per_class=1, \
                           n_repeated=0, n_redundant=0)

print(X.shape, y.shape)

df = pd.DataFrame(X,columns=['x1','x2'])
df['y'] = y

sns.scatterplot(data=df,x='x1',y='x2',hue='y')
```

```
(500, 2) (500,)
```

[2]:
```
<AxesSubplot:xlabel='x1', ylabel='x2'>
```



This is prety much it. linearly classifiable data with 2 classes.

11.1.1 Logistic Regression with Neural network (sigmoid)

logistic regression is like a linear classification model.

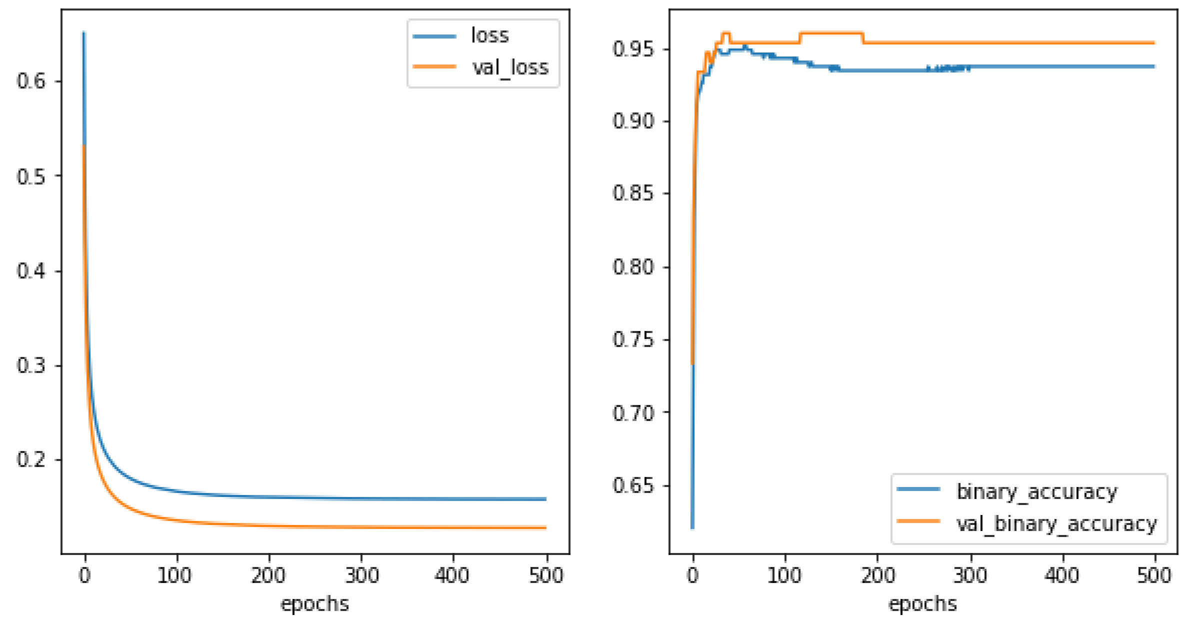
$$\begin{aligned} y &= g(h(x)) \\ h(x) &= wx + b \\ \text{sigmoid } g(x) &= \frac{1}{1+e^{-x}} \end{aligned}$$

So, actually after a linear weight and bias model, we need a sigmoid. It is actually called a perceptron.

[3]:
```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(units=1, activation='sigmoid')
])
model.compile(
    optimizer=tf.optimizers.SGD(learning_rate=0.09),
    loss=tf.keras.losses.BinaryCrossentropy(),
    metrics=tf.keras.metrics.BinaryAccuracy()
)

history = model.fit(
    df[['x1','x2']],
    df.y,
    epochs=500,
    batch_size=32,
    verbose=0,
    validation_split = 0.3)

history_metrics = pd.DataFrame(history.history)
history_metrics['epochs'] = history.epoch
```

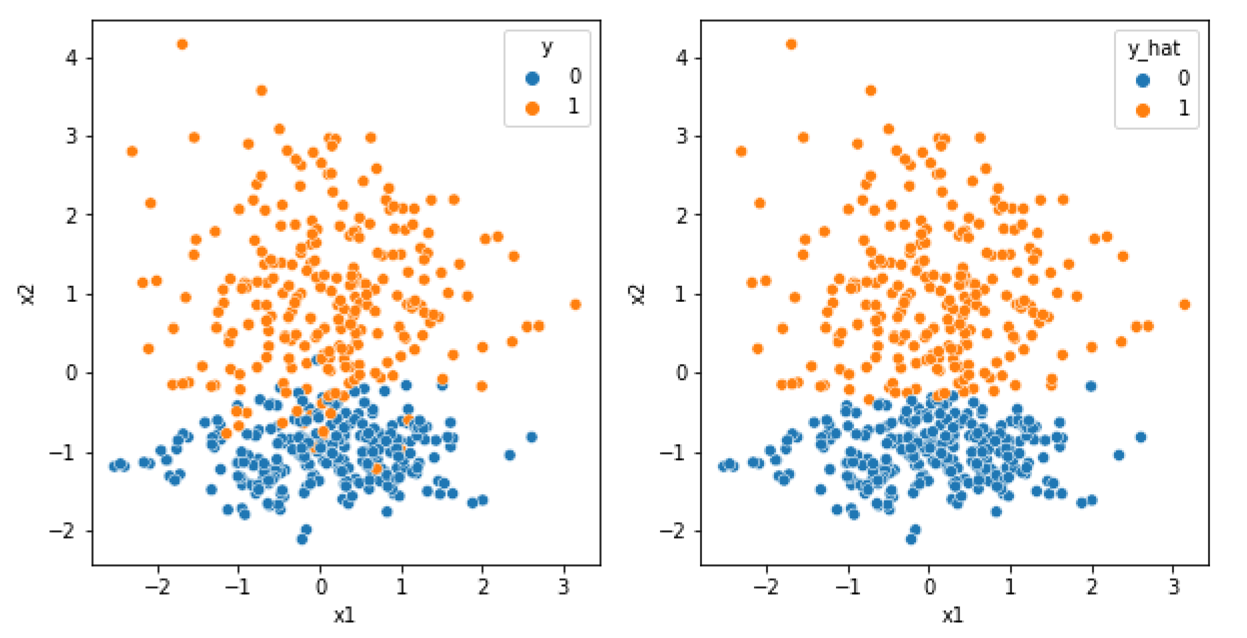
[4]:
```
fig,ax = plt.subplots(1,2,figsize=(10,5))
history_metrics.plot(x='epochs',y=['loss','val_loss'],ax=ax[0])
history_metrics.plot(x='epochs',y=['binary_accuracy','val_binary_accuracy'],ax=ax[1])
```

[4]:
```
<AxesSubplot:xlabel='epochs'>
```



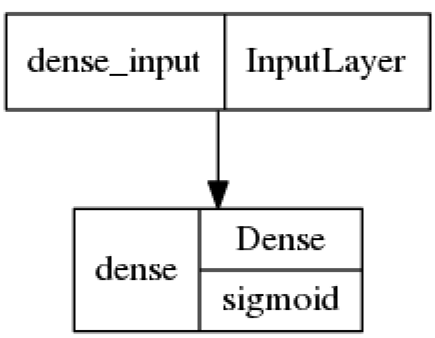
[5]:
```
y_hat = model.predict(df[['x1','x2']].values)
df['y_hat'] = np.array(y_hat>0.5,dtype='int')

fig,ax = plt.subplots(1,2,figsize=(10,5))
sns.scatterplot(data=df,x='x1',y='x2',hue='y',ax=ax[0])
sns.scatterplot(data=df,x='x1',y='x2',hue='y_hat', ax=ax[1])
```

[5]:
```
<AxesSubplot:xlabel='x1', ylabel='x2'>
```

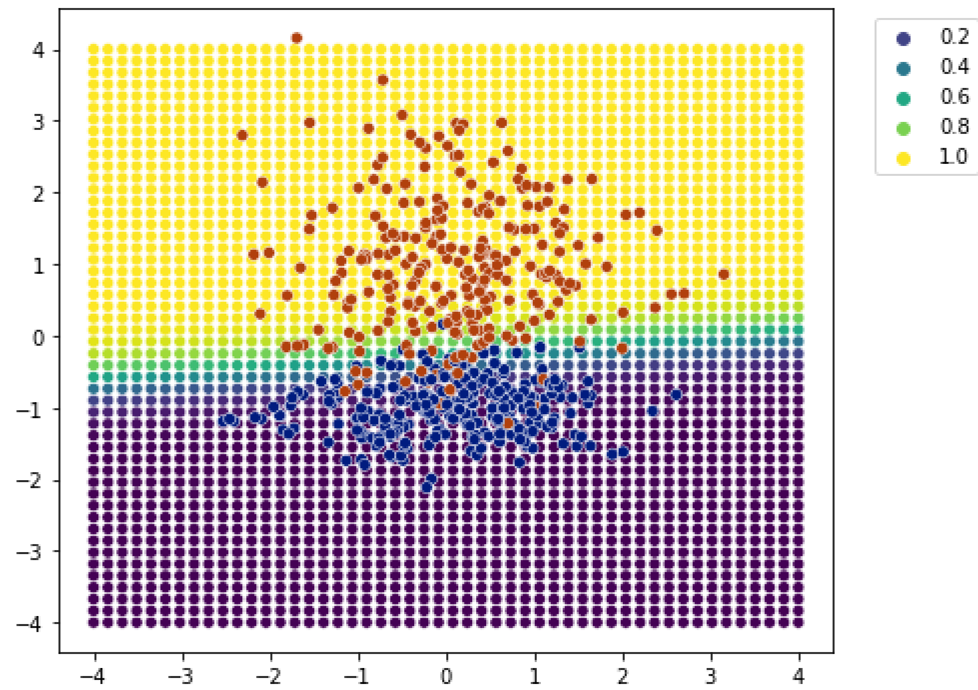


[6]:
```
tf.keras.utils.plot_model(model,show_layer_activations=True)
```

[6]:



[7]:
```
plot_classification_boundary(model.predict,data=df[['x1','x2','y']].values,size=4,
↪figsize=(7,5))
```



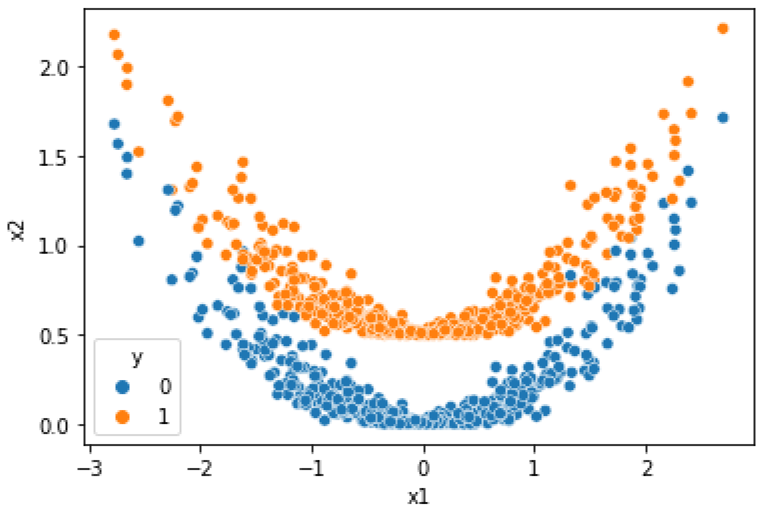
Looks to me that it worked out.

11.1.2 2 sigmoid layers

[8]:
```
x1, x2 = make_regression(n_features=1,noise=10, random_state=0, n_samples=500)

x2 = (x2/100)**2
```

[9]:
```
df1 = pd.DataFrame()
df1['x1'] = x1[...,-1]
df1['x2'] = x2
df1['y'] = 0

df2 = pd.DataFrame()
df2['x1'] = x1[...,-1]
df2['x2'] = x2 + 0.5
df2['y'] = 1


df = pd.concat((df1,df2)).sample(frac=1)

sns.scatterplot(data=df,x='x1',y='x2',hue='y')
```

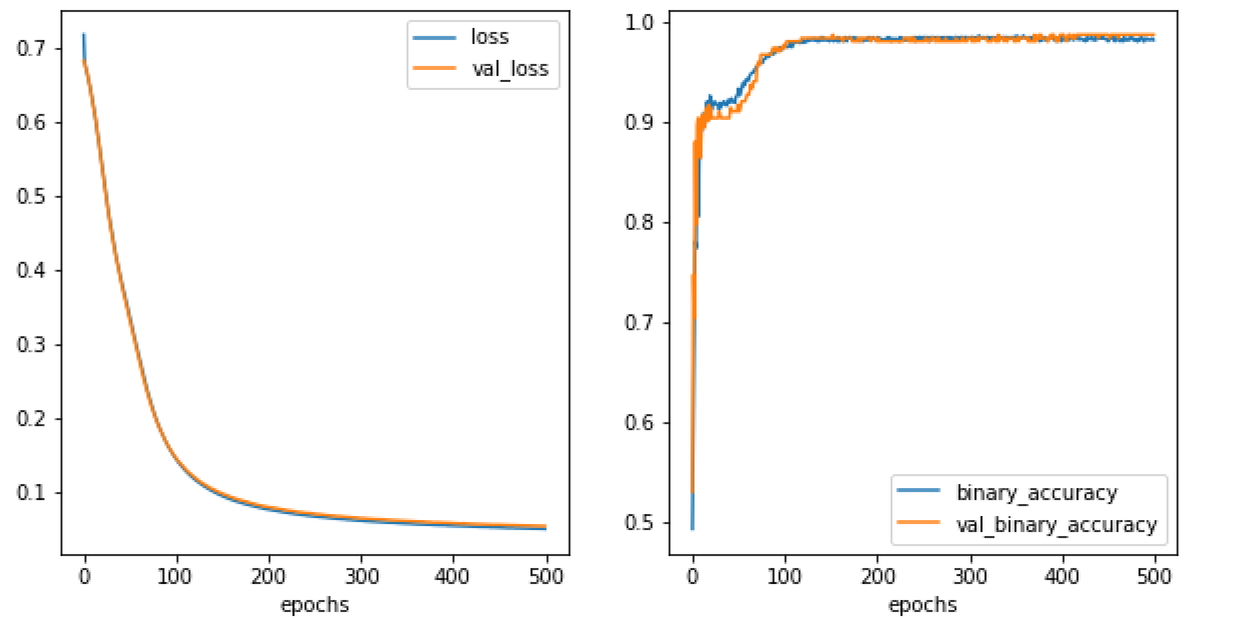
[9]:
```
<AxesSubplot:xlabel='x1', ylabel='x2'>
```



[10]:
```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(units=10, activation='sigmoid'),
    tf.keras.layers.Dense(units=1, activation='sigmoid')
])


model.compile(
    optimizer=tf.optimizers.SGD(learning_rate=0.1),
    loss=tf.keras.losses.BinaryCrossentropy(),
    metrics=tf.keras.metrics.BinaryAccuracy()
)

history = model.fit(
    df[['x1','x2']],
    df.y,
    epochs=500,
    batch_size=32,
    verbose=0,
    validation_split = 0.3)

history_metrics = pd.DataFrame(history.history)
history_metrics['epochs'] = history.epoch
```

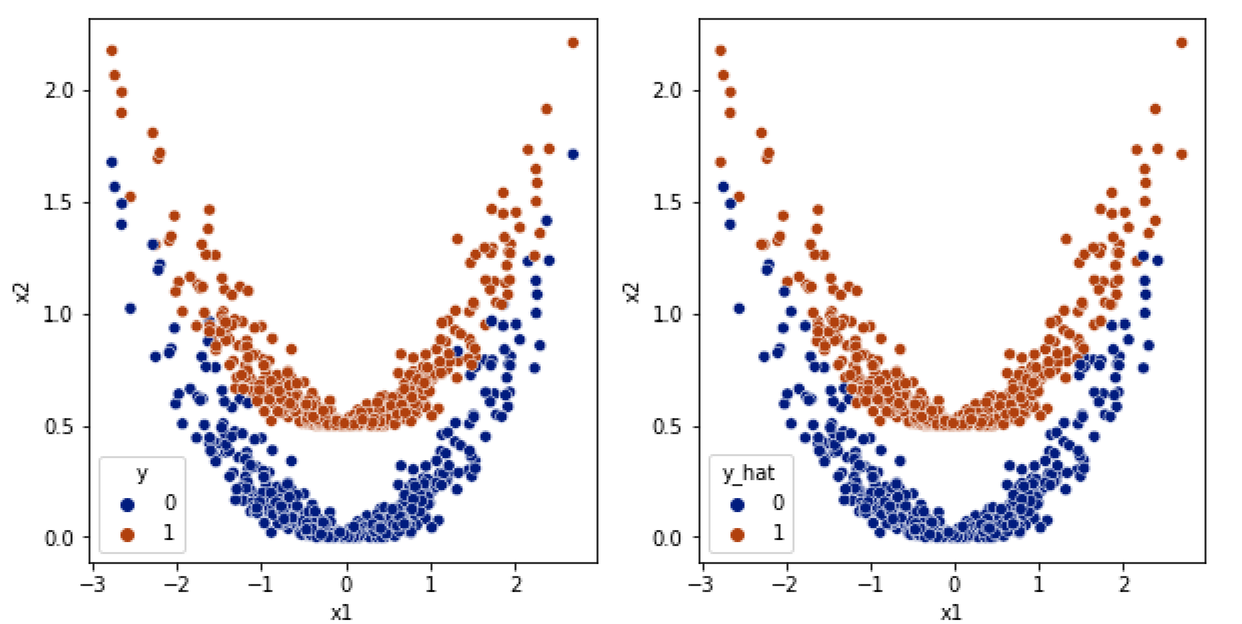
[11]:
```
fig,ax = plt.subplots(1,2,figsize=(10,5))

history_metrics.plot(x='epochs',y=['loss','val_loss'], ax=ax[0])
history_metrics.plot(x='epochs',y=['binary_accuracy','val_binary_accuracy'], ax=ax[1])
```

[11]:
```
<AxesSubplot:xlabel='epochs'>
```



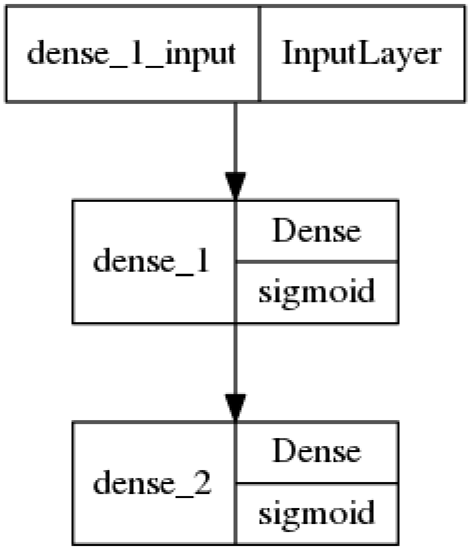
[12]:
```
y_hat = model.predict(df[['x1','x2']].values)
df['y_hat'] = np.array(y_hat>0.5,dtype='int')

fig,ax = plt.subplots(1,2,figsize=(10,5))
sns.scatterplot(data=df,x='x1',y='x2',hue='y',ax=ax[0], palette='dark')
sns.scatterplot(data=df,x='x1',y='x2',hue='y_hat', ax=ax[1], palette='dark')
```

[12]:
```
<AxesSubplot:xlabel='x1', ylabel='x2'>
```



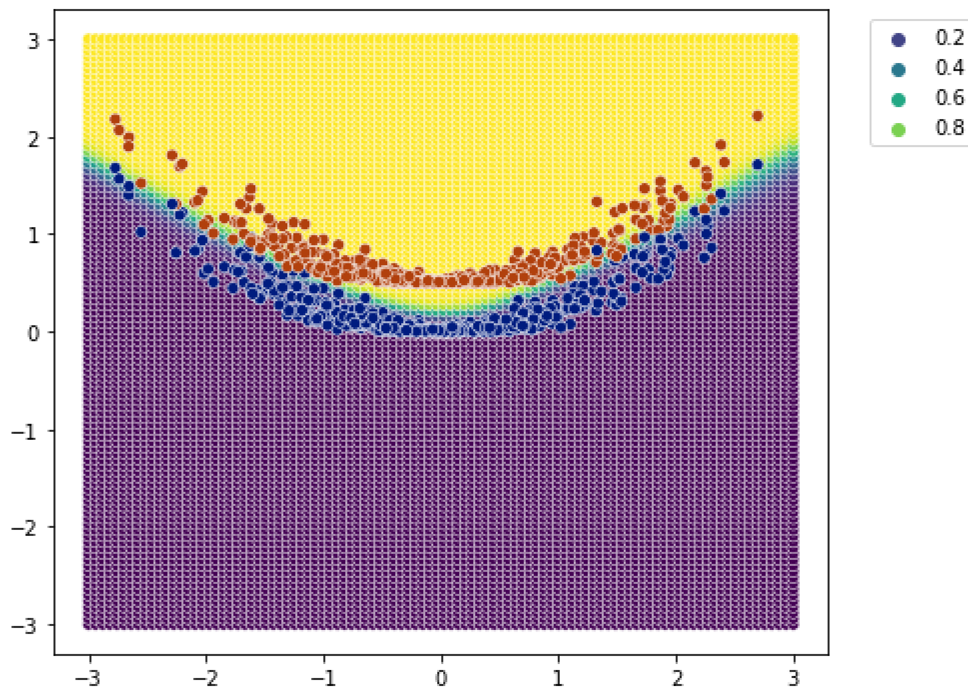
[13]:
```
tf.keras.utils.plot_model(model,show_layer_activations=True)
```

[13]:



[14]:

```
plot_classification_boundary(model.predict,data=df[['x1','x2','y']].values,size=3,\
                             figsize=(7,5), bound_details=100)
```

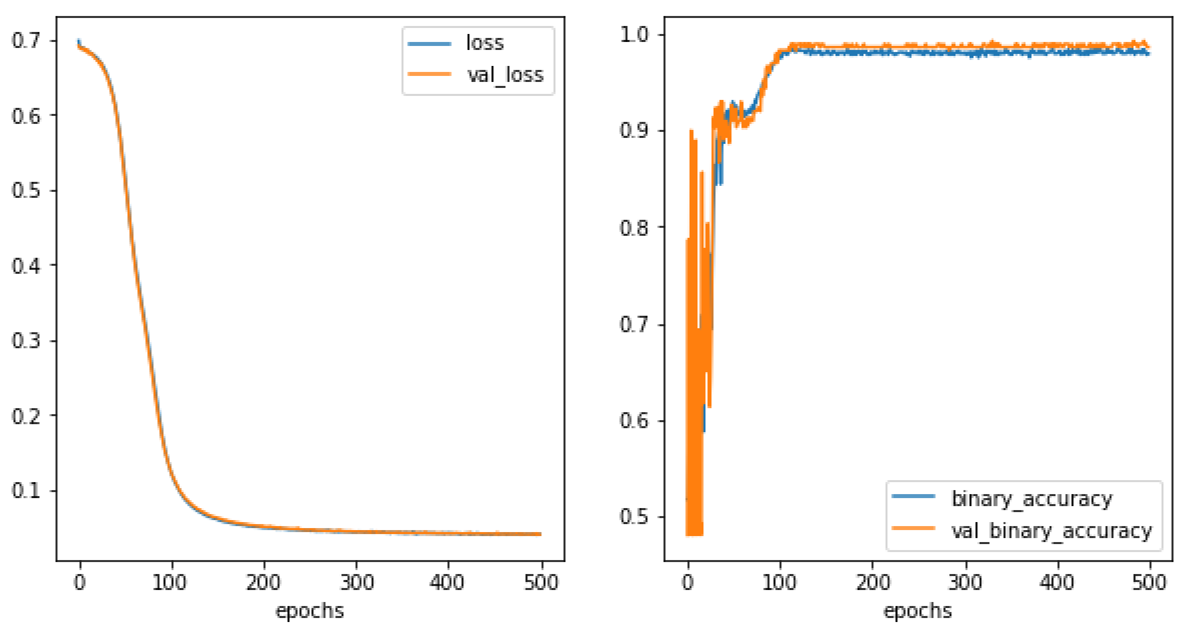


11.1.3 3 sigmoid layers

[15]:
```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(units=10, activation='sigmoid'),
    tf.keras.layers.Dense(units=10, activation='sigmoid'),
    tf.keras.layers.Dense(units=1, activation='sigmoid')
])


model.compile(
    optimizer=tf.optimizers.SGD(learning_rate=0.1),
    loss=tf.keras.losses.BinaryCrossentropy(),
    metrics=tf.keras.metrics.BinaryAccuracy()
)

history = model.fit(
    df[['x1','x2']],
    df.y,
    epochs=500,
    batch_size=32,
    verbose=0,
    validation_split = 0.3)

history_metrics = pd.DataFrame(history.history)
history_metrics['epochs'] = history.epoch
```

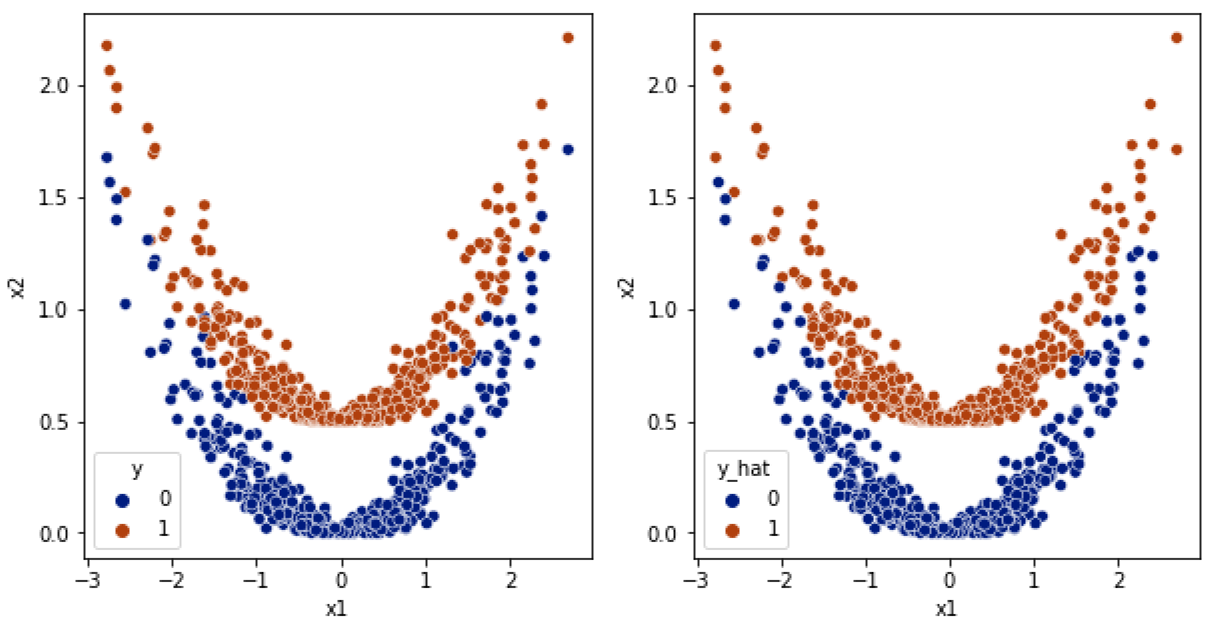
[16]:
```
fig,ax = plt.subplots(1,2,figsize=(10,5))
history_metrics.plot(x='epochs',y=['loss','val_loss'], ax=ax[0])
history_metrics.plot(x='epochs',y=['binary_accuracy','val_binary_accuracy'], ax=ax[1])
```

[16]:
```
<AxesSubplot:xlabel='epochs'>
```



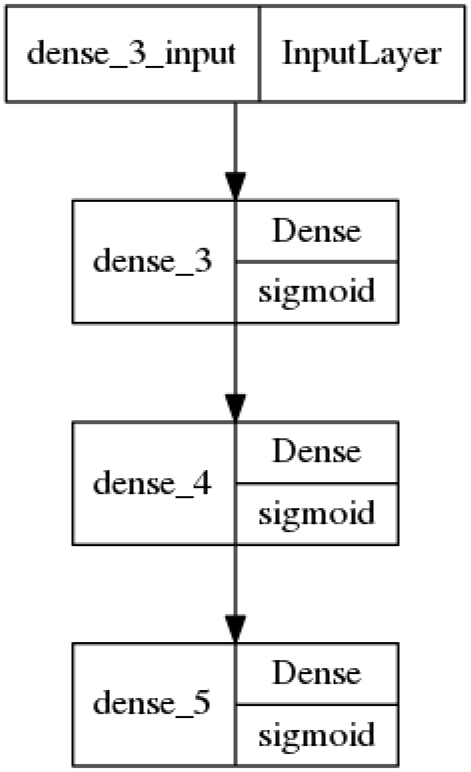
[17]:
```
y_hat = model.predict(df[['x1','x2']].values)
df['y_hat'] = np.array(y_hat>0.5,dtype='int')

fig,ax = plt.subplots(1,2,figsize=(10,5))
sns.scatterplot(data=df,x='x1',y='x2',hue='y',ax=ax[0], palette='dark')
sns.scatterplot(data=df,x='x1',y='x2',hue='y_hat', ax=ax[1], palette='dark')
```

[17]:
```
<AxesSubplot:xlabel='x1', ylabel='x2'>
```



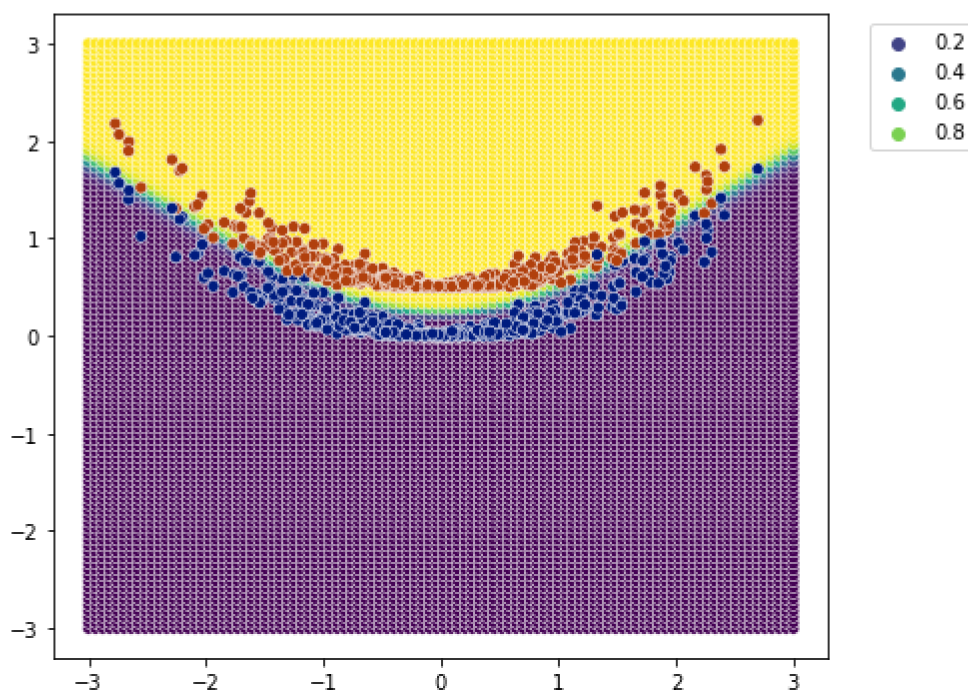
[18]:
```
tf.keras.utils.plot_model(model,show_layer_activations=True)
```

[18]:



[19]:

```
plot_classification_boundary(model.predict,data=df[['x1','x2','y']].values,size=3,\
                              figsize=(7,5), bound_details=100)
```



11.1.4 Something with relu and softmax

for using softmax i'll have to one hot encode the target.. so in the final layers we can have two outputs

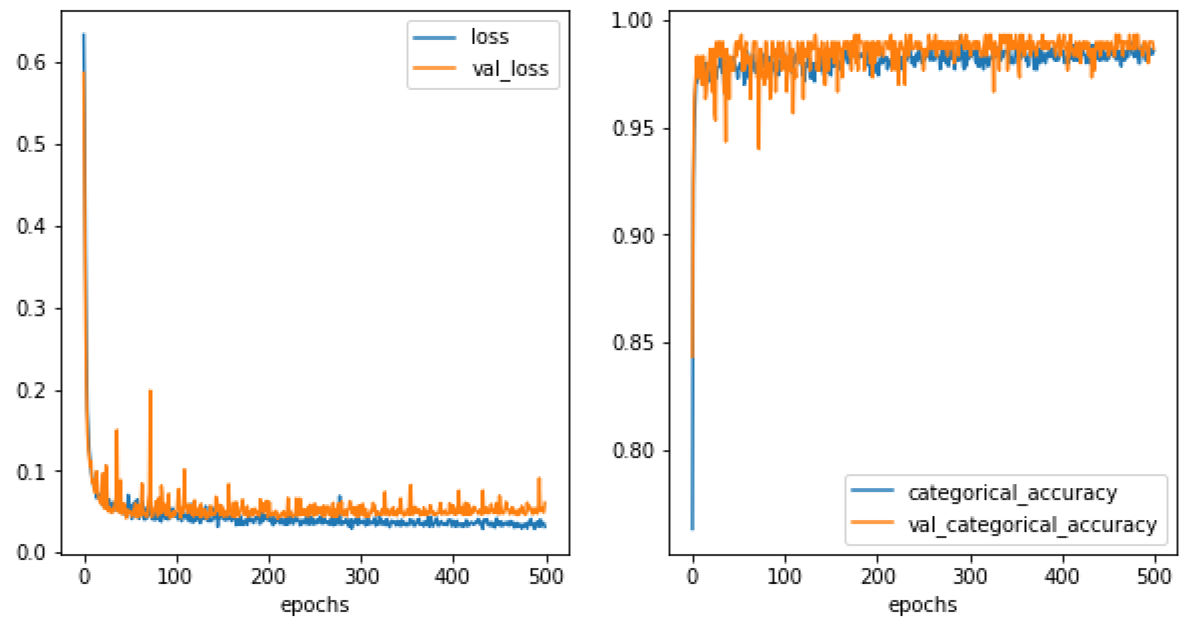
[20]:
```
from sklearn.preprocessing import OneHotEncoder
```

[21]:
```
ohe = OneHotEncoder()
ohe.fit(df[['y']].values)

y_ohe = ohe.transform(df[['y']].values).toarray()
```

[22]:
```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(units=10, activation='relu'),
    tf.keras.layers.Dense(units=10, activation='relu'),
    tf.keras.layers.Dense(units=2, activation='softmax')
])


model.compile(
    optimizer=tf.optimizers.SGD(learning_rate=0.1),
    loss=tf.keras.losses.CategoricalCrossentropy(),
    metrics=tf.keras.metrics.CategoricalAccuracy()
)

history = model.fit(
    df[['x1','x2']],
    y_ohe,
    epochs=500,
    batch_size=32,
    verbose=0,
    validation_split = 0.3
)
```

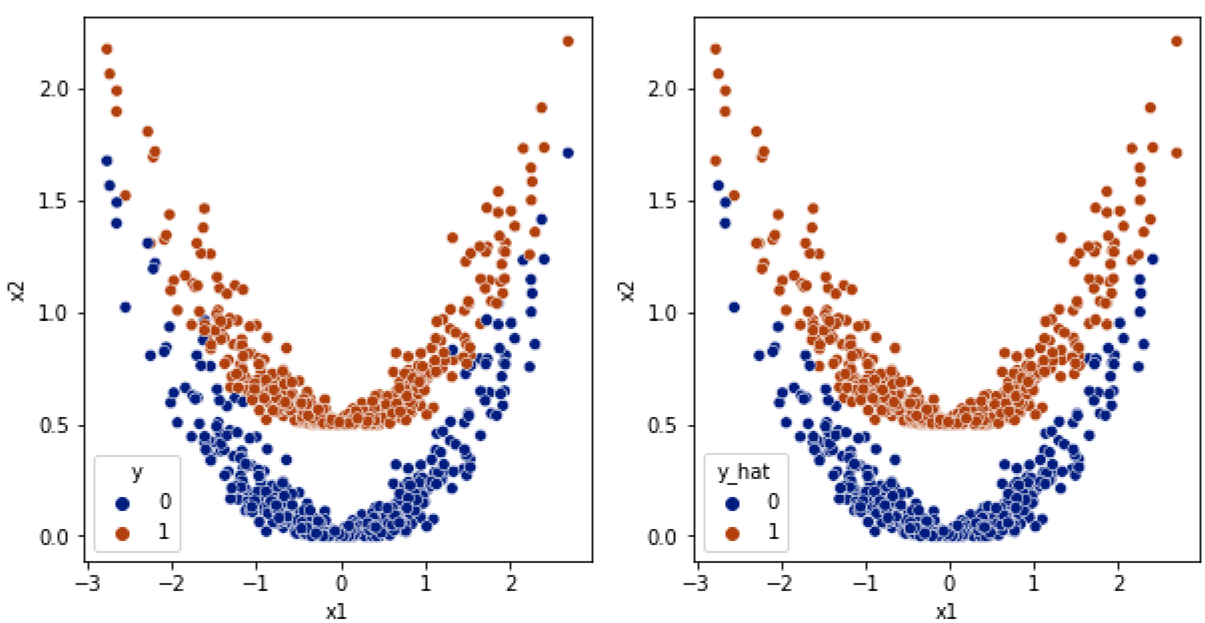
[23]:
```
history_metrics = pd.DataFrame(history.history)
history_metrics['epochs'] = history.epoch

fig,ax = plt.subplots(1,2,figsize=(10,5))

history_metrics.plot(x='epochs',y=['loss','val_loss'], ax=ax[0])
history_metrics.plot(x='epochs',y=['categorical_accuracy','val_categorical_accuracy'],␣
↪ax=ax[1])
```

[23]:
```
<AxesSubplot:xlabel='epochs'>
```



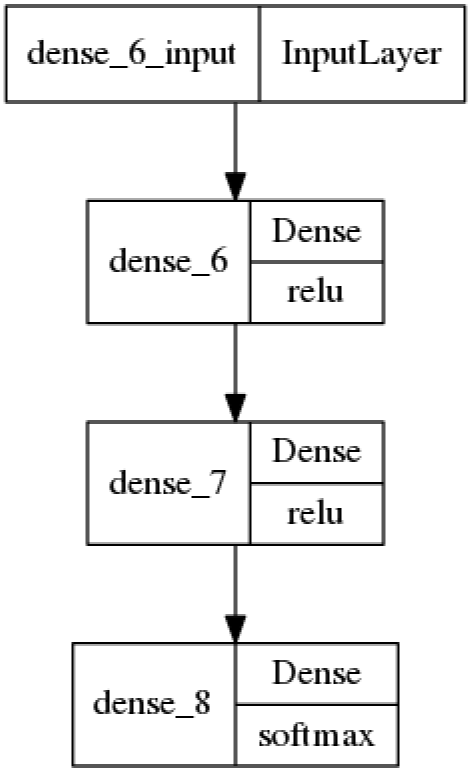
[24]:
```
y_hat = np.argmax(model.predict(df[['x1','x2']].values),axis=1)

df['y_hat'] = y_hat

fig,ax = plt.subplots(1,2,figsize=(10,5))
sns.scatterplot(data=df,x='x1',y='x2',hue='y',ax=ax[0], palette='dark')
sns.scatterplot(data=df,x='x1',y='x2',hue='y_hat', ax=ax[1], palette='dark')
```

[24]:
```
<AxesSubplot:xlabel='x1', ylabel='x2'>
```



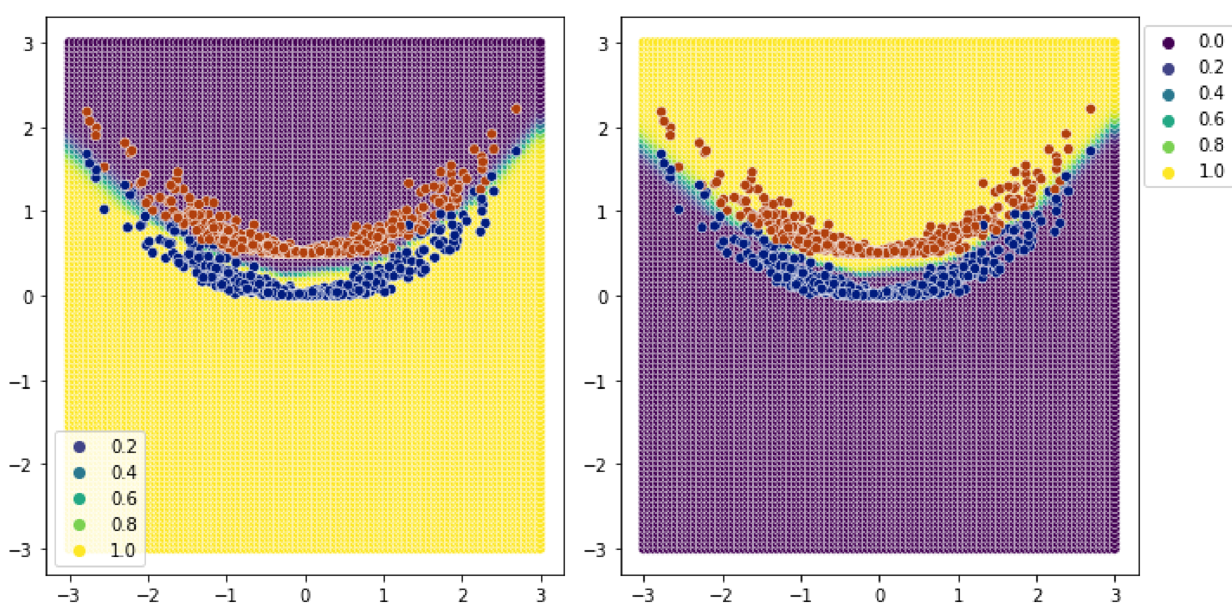
[25]:
```
tf.keras.utils.plot_model(model,show_layer_activations=True)
```

[25]:



[26]:

```
plot_classification_boundary(model.predict,data=df[['x1','x2','y']].values,size=3,\
                              figsize=(10,5), bound_details=100, n_plot_cols=2)
```



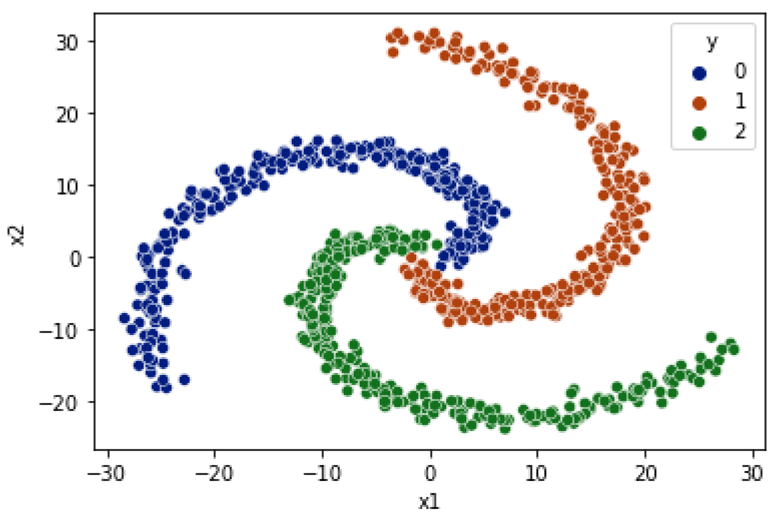
11.2 Easy Spiral Classification

[27]:
```
from mightypy.ml.dataset import generate_spiral_data
```

[28]:
```
data_limit = 30

X, y = generate_spiral_data(data_limit=data_limit, n_classes=3)
```

[29]:
```
df = pd.DataFrame(data=X, columns=['x1','x2'])
df['y'] = y

df = df.sample(frac=1)

sns.scatterplot(data=df, x='x1',y='x2',hue='y',palette='dark')
```

[29]:
```
<AxesSubplot:xlabel='x1', ylabel='x2'>
```



[30]:
```
ohe = OneHotEncoder()
ohe.fit(df[['y']].values)

y_ohe = ohe.transform(df[['y']].values).toarray()
```

11.2.1 1 sigmoid

[31]:
```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(units=64, activation='sigmoid'),
    tf.keras.layers.Dense(units=3, activation='softmax')
])


model.compile(
    optimizer=tf.optimizers.SGD(learning_rate=1e-2),
    loss=tf.keras.losses.CategoricalCrossentropy(),
```

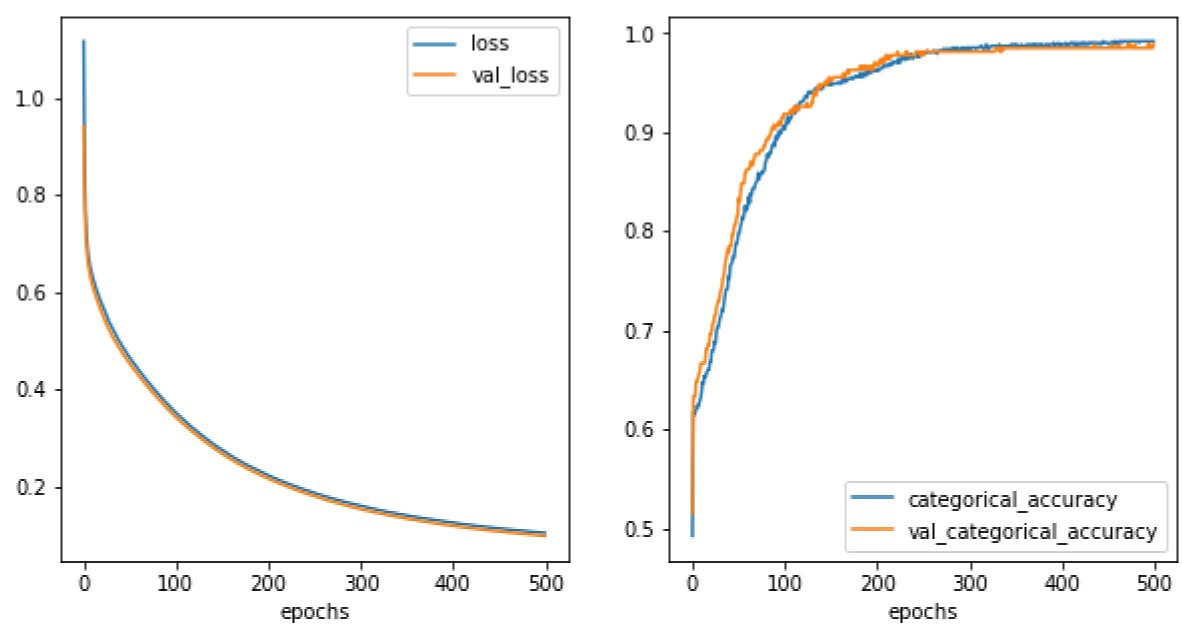
(continues on next page)

(continued from previous page)

```
    metrics=tf.keras.metrics.CategoricalAccuracy()
)

history = model.fit(
    df[['x1','x2']],
    y_ohe,
    epochs=500,
    batch_size=32,
    verbose=0,
    validation_split = 0.3
)

history_metrics = pd.DataFrame(history.history)
history_metrics['epochs'] = history.epoch
```

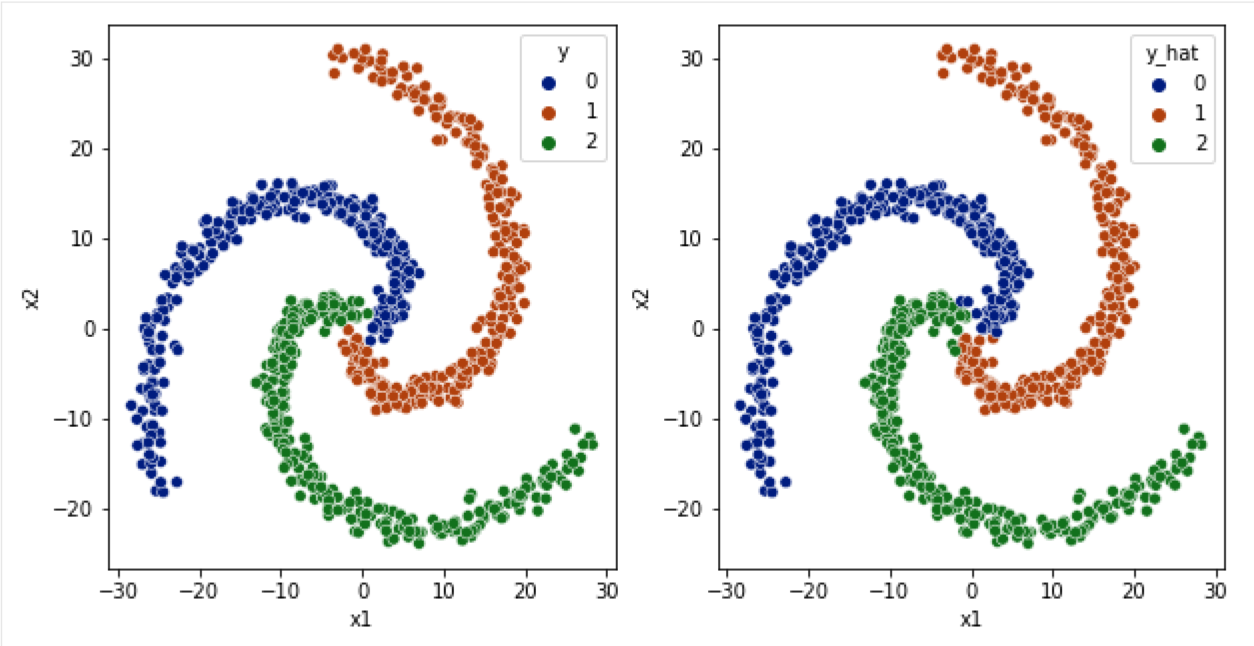
[32]:
```
fig,ax = plt.subplots(1,2,figsize=(10,5))

history_metrics.plot(x='epochs',y=['loss','val_loss'], ax=ax[0])
history_metrics.plot(x='epochs',y=['categorical_accuracy','val_categorical_accuracy'],
↪ax=ax[1])
```

[32]:
```
<AxesSubplot:xlabel='epochs'>
```

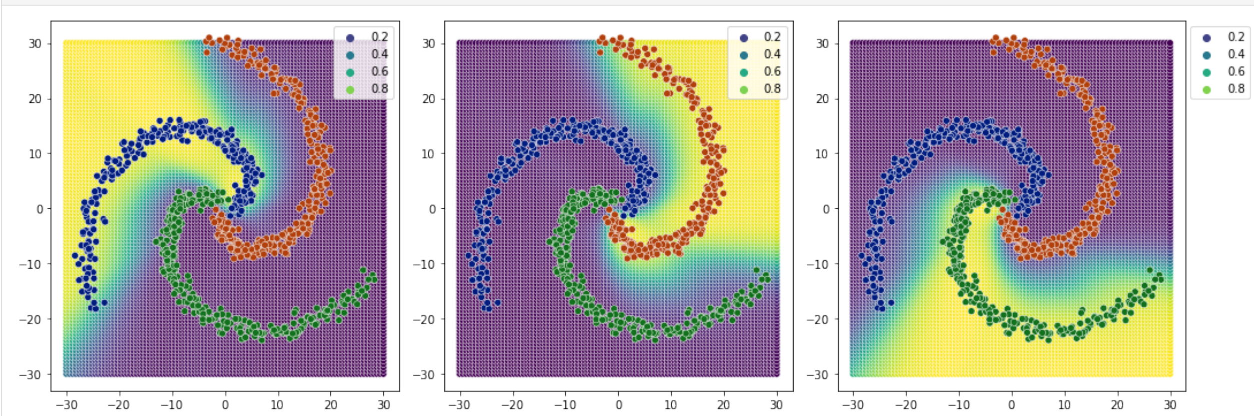


[33]:
```
df['y_hat'] = np.argmax(model.predict(df[['x1','x2']].values), axis=1)

fig,ax = plt.subplots(1,2,figsize=(10,5))
sns.scatterplot(data=df,x='x1',y='x2',hue='y',ax=ax[0], palette='dark')
sns.scatterplot(data=df,x='x1',y='x2',hue='y_hat', ax=ax[1], palette='dark')
```

[33]:
```
<AxesSubplot:xlabel='x1', ylabel='x2'>
```




```
[34]: plot_classification_boundary(model.predict,data=df[['x1','x2','y']].values,size=30,\
                                     figsize=(15,5), bound_details=100, n_plot_cols=3)
```



11.2.2 2 sigmoids

```
[35]: model = tf.keras.Sequential([
          tf.keras.layers.Dense(units=64, activation='sigmoid'),
          tf.keras.layers.Dense(units=64, activation='sigmoid'),
          tf.keras.layers.Dense(units=3, activation='softmax')
      ])


      model.compile(
          optimizer=tf.optimizers.SGD(learning_rate=1e-2),
          loss=tf.keras.losses.CategoricalCrossentropy(),
          metrics=tf.keras.metrics.CategoricalAccuracy()
      )
```

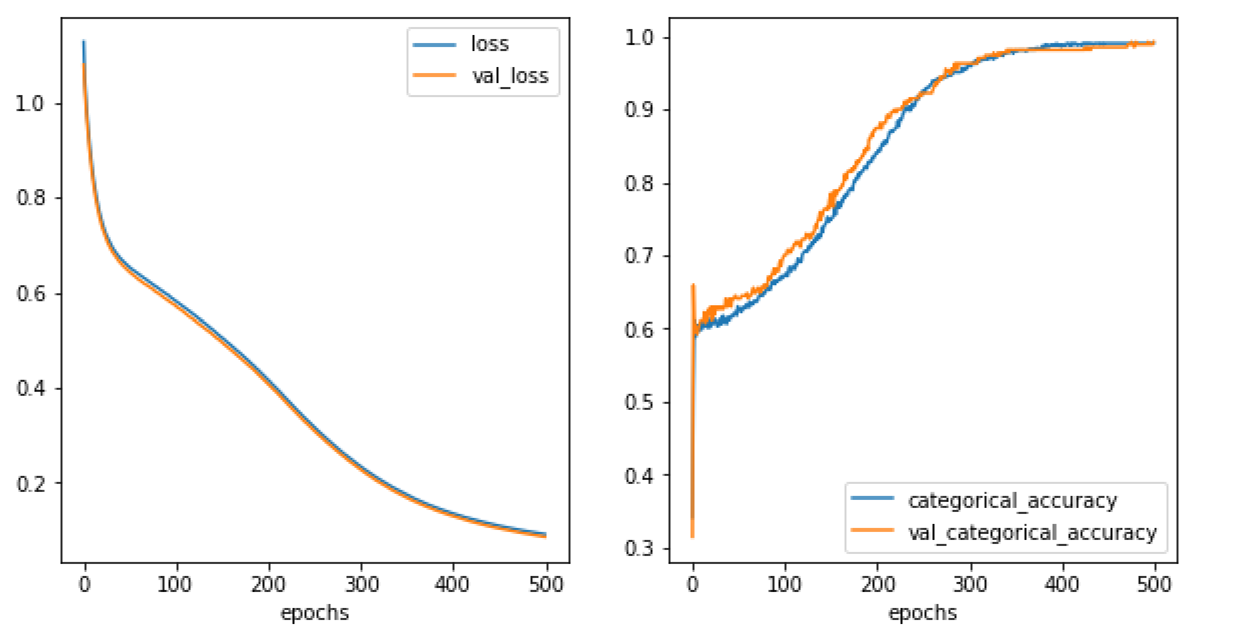
(continues on next page)

(continued from previous page)

```
history = model.fit(
    df[['x1','x2']],
    y_ohe,
    epochs=500,
    batch_size=32,
    verbose=0,
    validation_split = 0.3
)

history_metrics = pd.DataFrame(history.history)
history_metrics['epochs'] = history.epoch
```

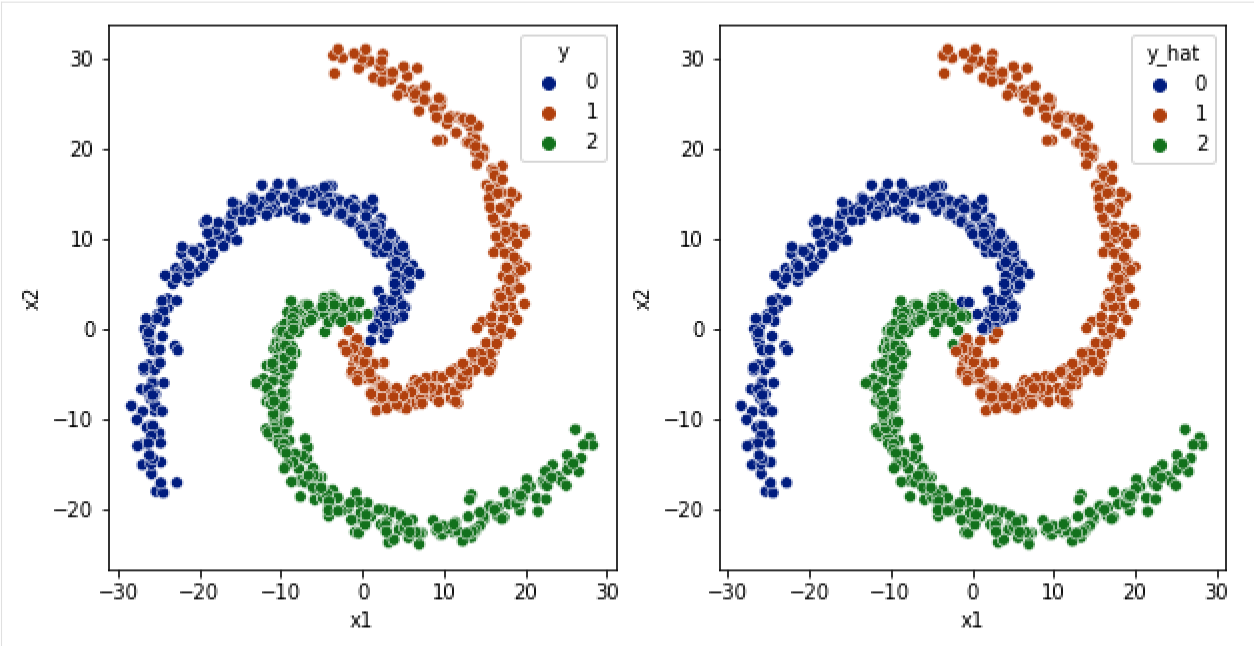
[36]:
```
fig,ax = plt.subplots(1,2,figsize=(10,5))
history_metrics.plot(x='epochs',y=['loss','val_loss'], ax=ax[0])
history_metrics.plot(x='epochs',y=['categorical_accuracy','val_categorical_accuracy'], ax=ax[1])
```

[36]:
```
<AxesSubplot:xlabel='epochs'>
```

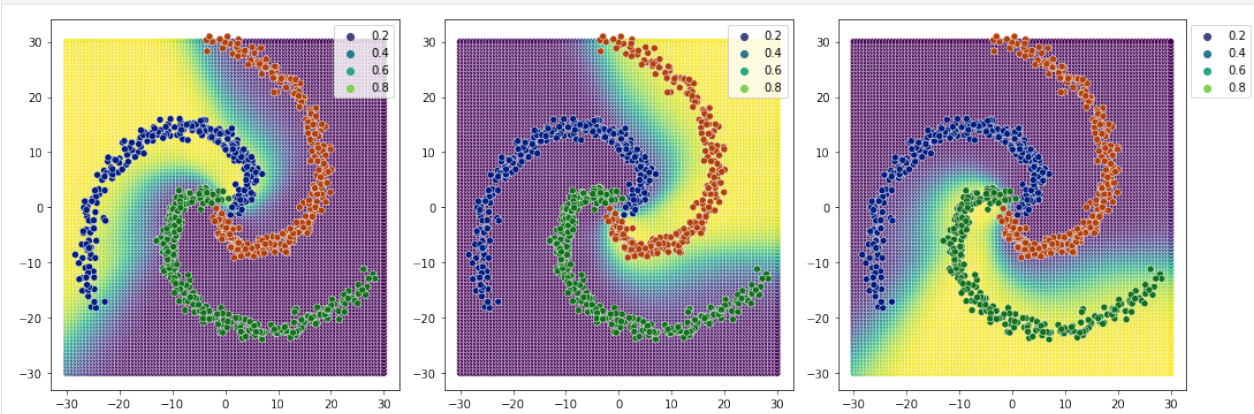


[37]:
```
df['y_hat'] = np.argmax(model.predict(df[['x1','x2']].values), axis=1)

fig,ax = plt.subplots(1,2,figsize=(10,5))
sns.scatterplot(data=df,x='x1',y='x2',hue='y',ax=ax[0], palette='dark')
sns.scatterplot(data=df,x='x1',y='x2',hue='y_hat', ax=ax[1], palette='dark')
```

[37]:
```
<AxesSubplot:xlabel='x1', ylabel='x2'>
```



[38]:
```
plot_classification_boundary(model.predict,data=df[['x1','x2','y']].values,size=30,\
                              figsize=(15,5), bound_details=100, n_plot_cols=3)
```



11.2.3 1 relu layer

[39]:
```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(units=64, activation='relu'),
    tf.keras.layers.Dense(units=3, activation='softmax')
])


model.compile(
    optimizer=tf.optimizers.Adam(learning_rate=1e-2),
    loss=tf.keras.losses.CategoricalCrossentropy(),
    metrics=tf.keras.metrics.CategoricalAccuracy()
)
```

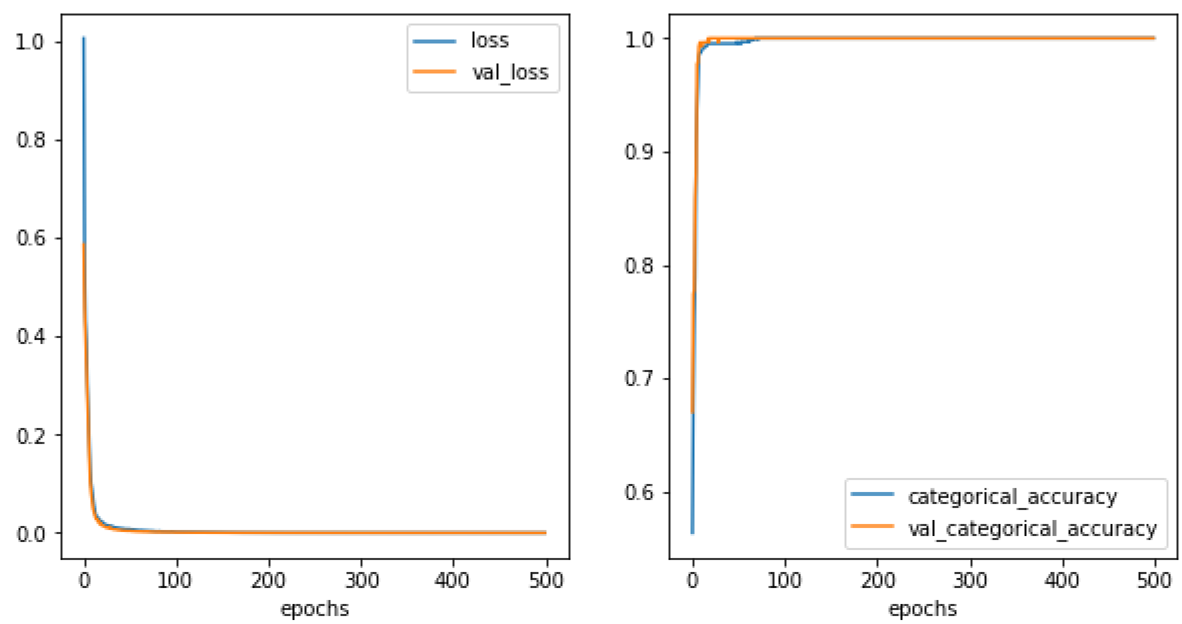
(continues on next page)

(continued from previous page)

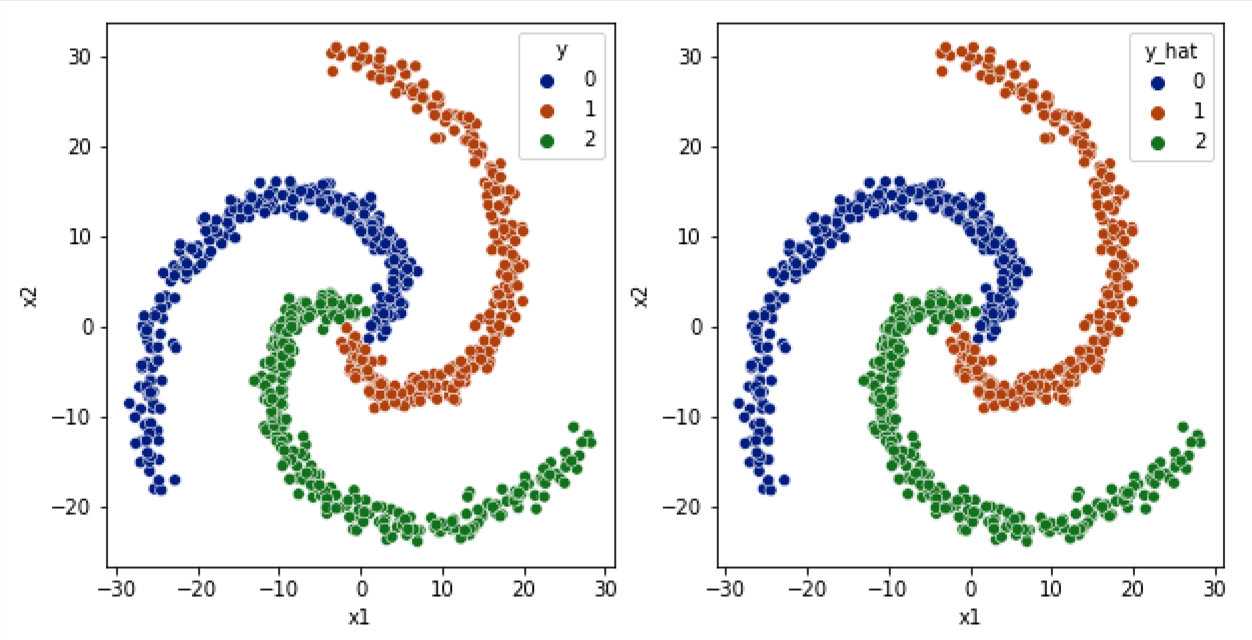
```
history = model.fit(
    df[['x1','x2']],
    y_ohe,
    epochs=500,
    batch_size=32,
    verbose=0,
    validation_split = 0.3
)

history_metrics = pd.DataFrame(history.history)
history_metrics['epochs'] = history.epoch
```

[40]:
```
fig,ax = plt.subplots(1,2,figsize=(10,5))
history_metrics.plot(x='epochs',y=['loss','val_loss'], ax=ax[0])
history_metrics.plot(x='epochs',y=['categorical_accuracy','val_categorical_accuracy'],␣
↪ax=ax[1])
```

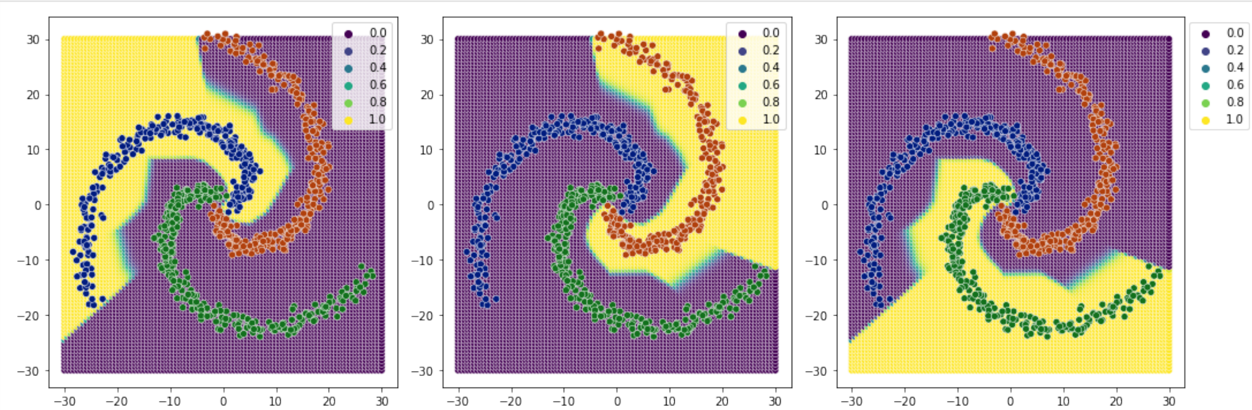
[40]:
```
<AxesSubplot:xlabel='epochs'>
```



[41]:
```
df['y_hat'] = np.argmax(model.predict(df[['x1','x2']].values), axis=1)

fig,ax = plt.subplots(1,2,figsize=(10,5))
sns.scatterplot(data=df,x='x1',y='x2',hue='y',ax=ax[0], palette='dark')
sns.scatterplot(data=df,x='x1',y='x2',hue='y_hat', ax=ax[1], palette='dark')
plt.show()
```



[42]:
```
plot_classification_boundary(model.predict,data=df[['x1','x2','y']].values,size=30,\
                                        figsize=(15,5), bound_details=100, n_plot_cols=3)
```



relu as compared to sigmoid, creates sharper edges(more clear probabilities).

11.2.4 2 relu layers

[43]:
```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(units=100, activation='relu'),
    tf.keras.layers.Dense(units=100, activation='relu'),
    tf.keras.layers.Dense(units=3, activation='softmax')
])


model.compile(
    optimizer=tf.optimizers.Adam(learning_rate=0.01),
    loss=tf.keras.losses.CategoricalCrossentropy(),
```

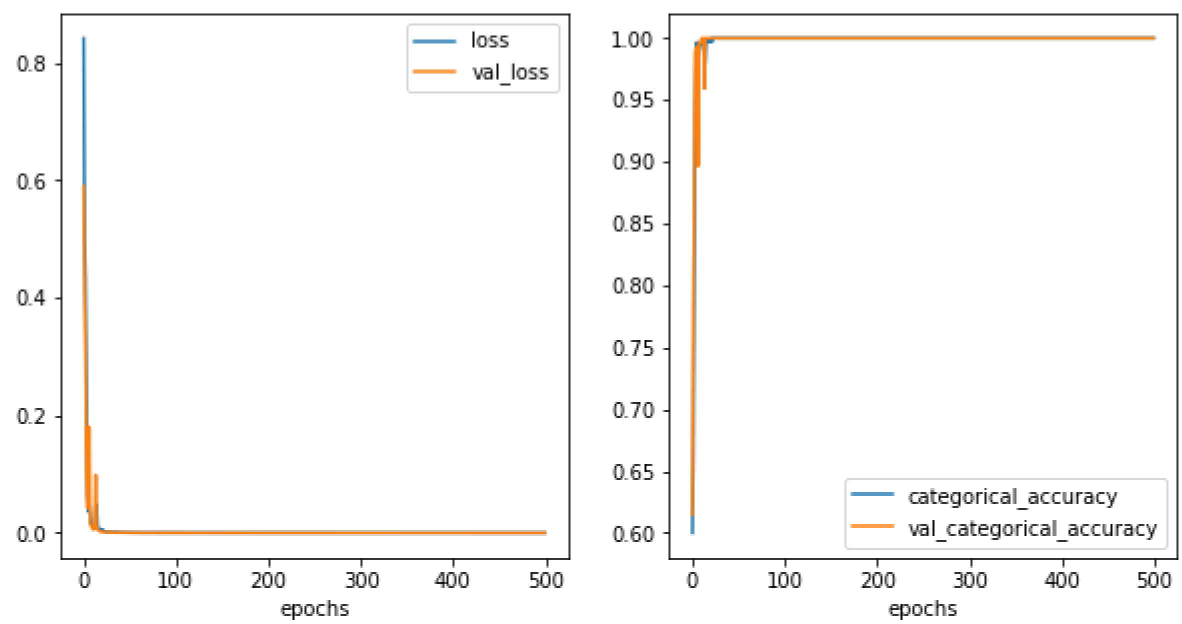
(continues on next page)

(continued from previous page)

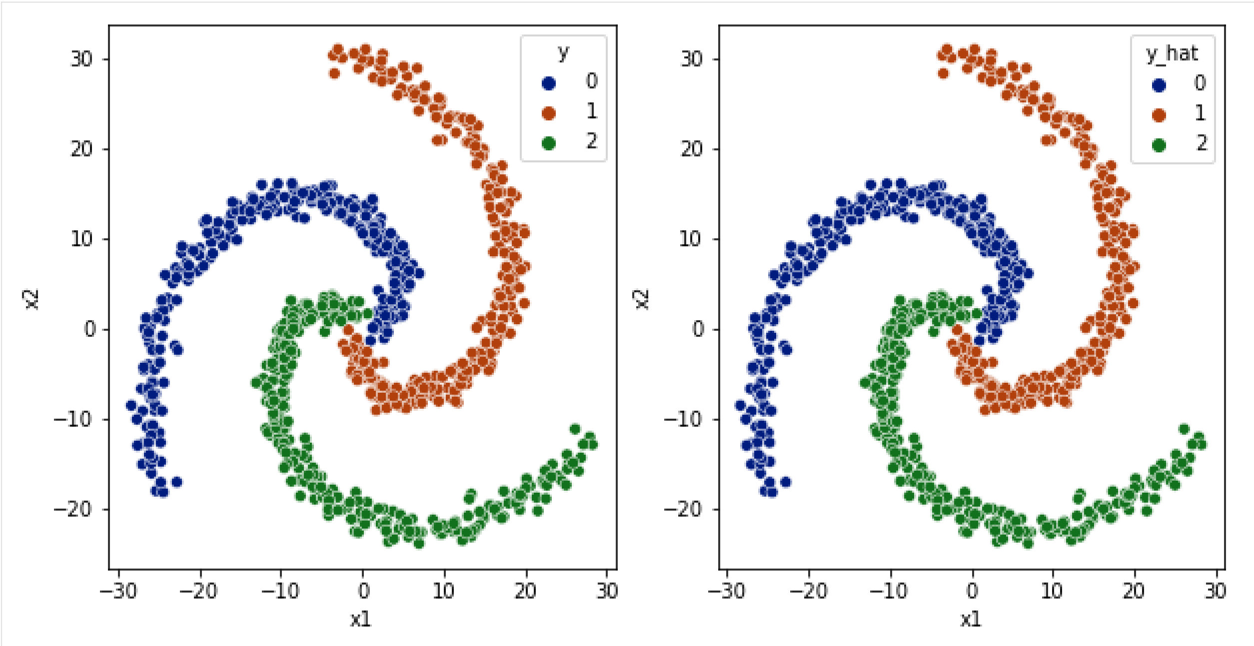
```
    metrics=tf.keras.metrics.CategoricalAccuracy()
)

history = model.fit(
    df[['x1','x2']],
    y_ohe,
    epochs=500,
    batch_size=32,
    verbose=0,
    validation_split = 0.3
)

history_metrics = pd.DataFrame(history.history)
history_metrics['epochs'] = history.epoch
```

[44]:
```
fig,ax = plt.subplots(1,2,figsize=(10,5))
history_metrics.plot(x='epochs',y=['loss','val_loss'], ax=ax[0])
history_metrics.plot(x='epochs',y=['categorical_accuracy','val_categorical_accuracy'],␣
↪ax=ax[1])
```

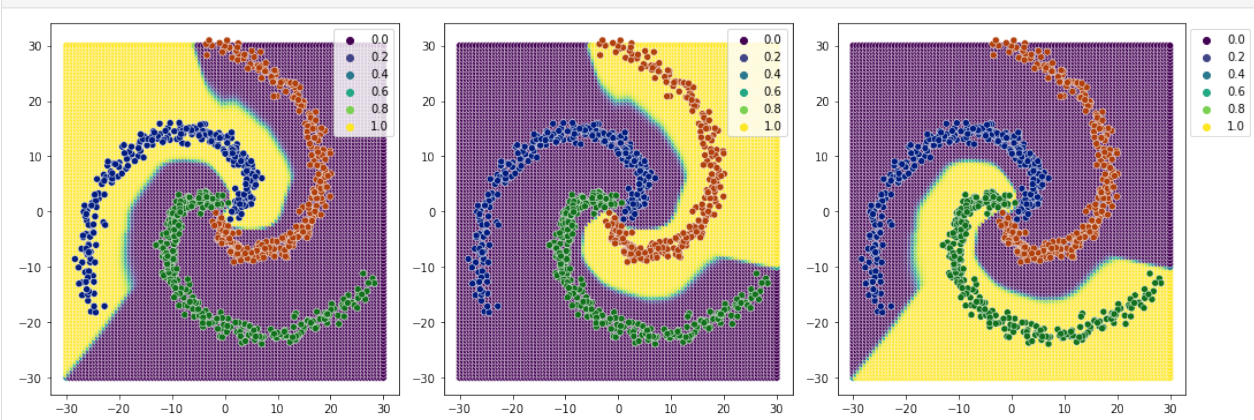
[44]:
```
<AxesSubplot:xlabel='epochs'>
```



[45]:
```
df['y_hat'] = np.argmax(model.predict(df[['x1','x2']].values), axis=1)

fig,ax = plt.subplots(1,2,figsize=(10,5))
sns.scatterplot(data=df,x='x1',y='x2',hue='y',ax=ax[0], palette='dark')
sns.scatterplot(data=df,x='x1',y='x2',hue='y_hat', ax=ax[1], palette='dark')
plt.show()
```




```
[46]: plot_classification_boundary(model.predict,data=df[['x1','x2','y']].values,size=30,\
                                   figsize=(15,5), bound_details=100, n_plot_cols=3)
```

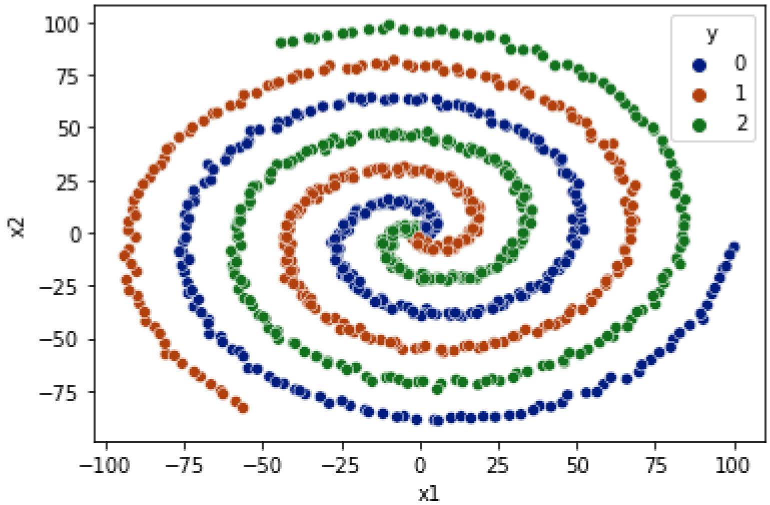


1 layer relu almost did the job. 2 layer relu is almost the same.

11.3 Complex Spiral Classification

```
[47]: data_limit = 100

X, y = generate_spiral_data(data_limit=data_limit, n_classes=3)

df = pd.DataFrame(data=X, columns=['x1','x2'])
df['y'] = y

df = df.sample(frac=1)

sns.scatterplot(data=df, x='x1',y='x2',hue='y',palette='dark')
```

[47]: <AxesSubplot:xlabel='x1', ylabel='x2'>



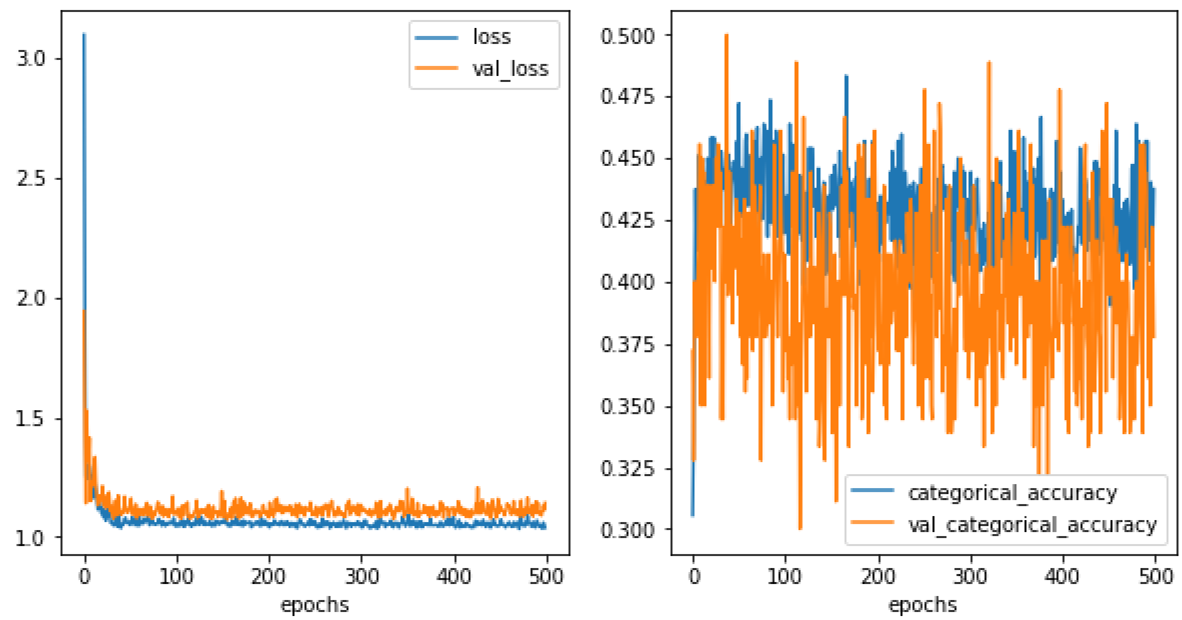
[48]:
```
ohe = OneHotEncoder()
ohe.fit(df[['y']].values)

y_ohe = ohe.transform(df[['y']].values).toarray()
```

11.3.1 1 relu layer

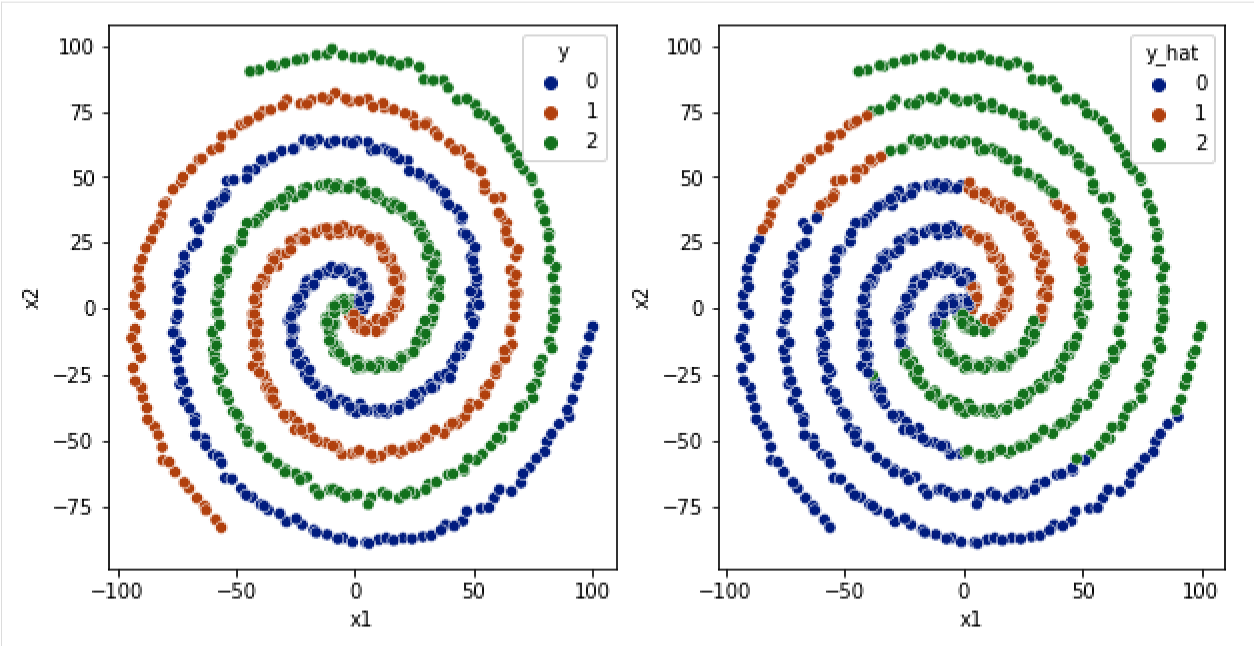
[49]:
```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(units=100, activation='relu'),
    tf.keras.layers.Dense(units=3, activation='softmax')
])


model.compile(
    optimizer=tf.optimizers.Adam(learning_rate=0.01),
    loss=tf.keras.losses.CategoricalCrossentropy(),
    metrics=tf.keras.metrics.CategoricalAccuracy()
)

history = model.fit(
    df[['x1','x2']],
    y_ohe,
    epochs=500,
    batch_size=32,
    verbose=0,
    validation_split = 0.2
)

history_metrics = pd.DataFrame(history.history)
history_metrics['epochs'] = history.epoch
```

[50]:
```
fig,ax = plt.subplots(1,2,figsize=(10,5))

history_metrics.plot(x='epochs',y=['loss','val_loss'], ax=ax[0])
history_metrics.plot(x='epochs',y=['categorical_accuracy','val_categorical_accuracy'],␣
↪ax=ax[1])
```

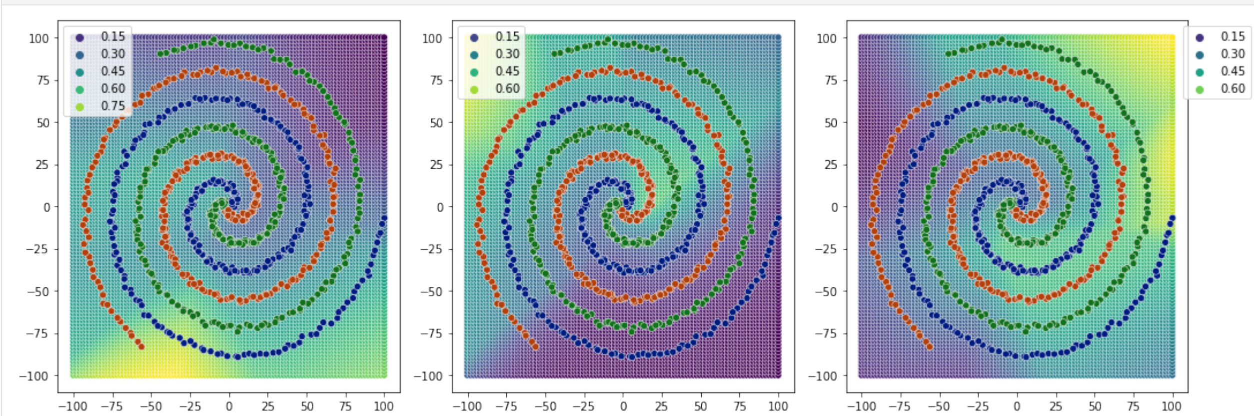
[50]:
```
<AxesSubplot:xlabel='epochs'>
```



[51]:
```
df['y_hat'] = np.argmax(model.predict(df[['x1','x2']].values), axis=1)

fig,ax = plt.subplots(1,2,figsize=(10,5))
sns.scatterplot(data=df,x='x1',y='x2',hue='y',ax=ax[0], palette='dark')
sns.scatterplot(data=df,x='x1',y='x2',hue='y_hat', ax=ax[1], palette='dark')
plt.show()
```



[52]:
```
plot_classification_boundary(model.predict,data=df[['x1','x2','y']].values,size=data_
↪limit,\
                                        figsize=(15,5), bound_details=100, n_plot_cols=3)
```



So, for 1 relu layer, it is not able to converge. and decision boundaries are not very clear.

11.3.2 2 relu layers

[53]:
```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(units=100, activation='relu'),
    tf.keras.layers.Dense(units=100, activation='relu'),
    tf.keras.layers.Dense(units=3, activation='softmax')
])


model.compile(
    optimizer=tf.optimizers.Adam(learning_rate=0.001),
```

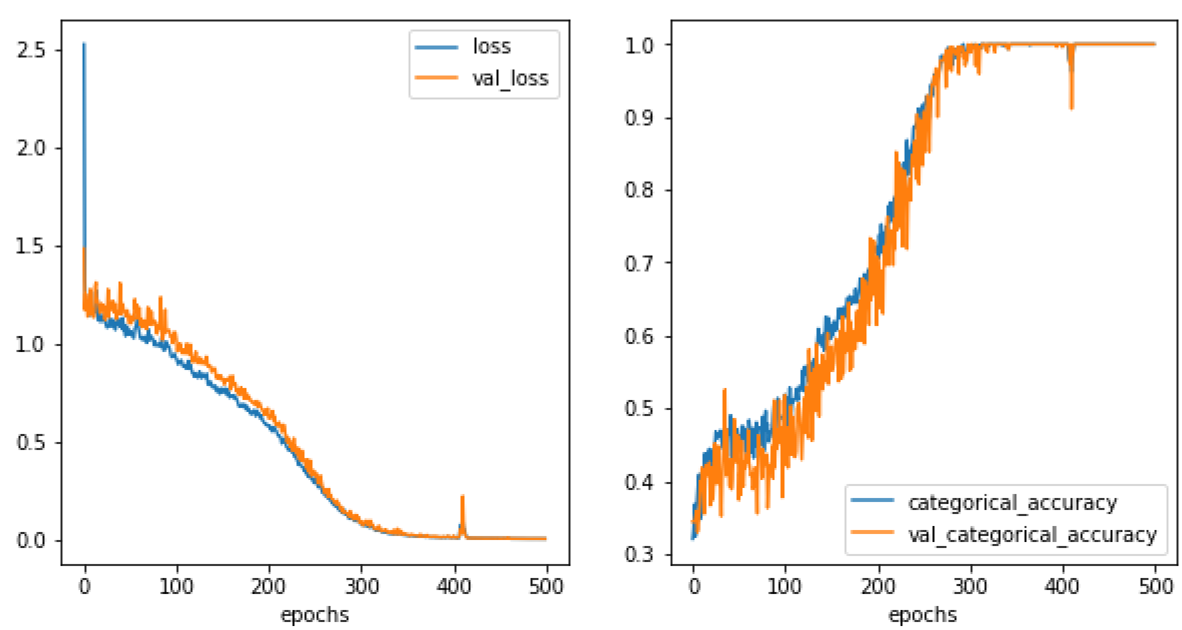
(continues on next page)

(continued from previous page)

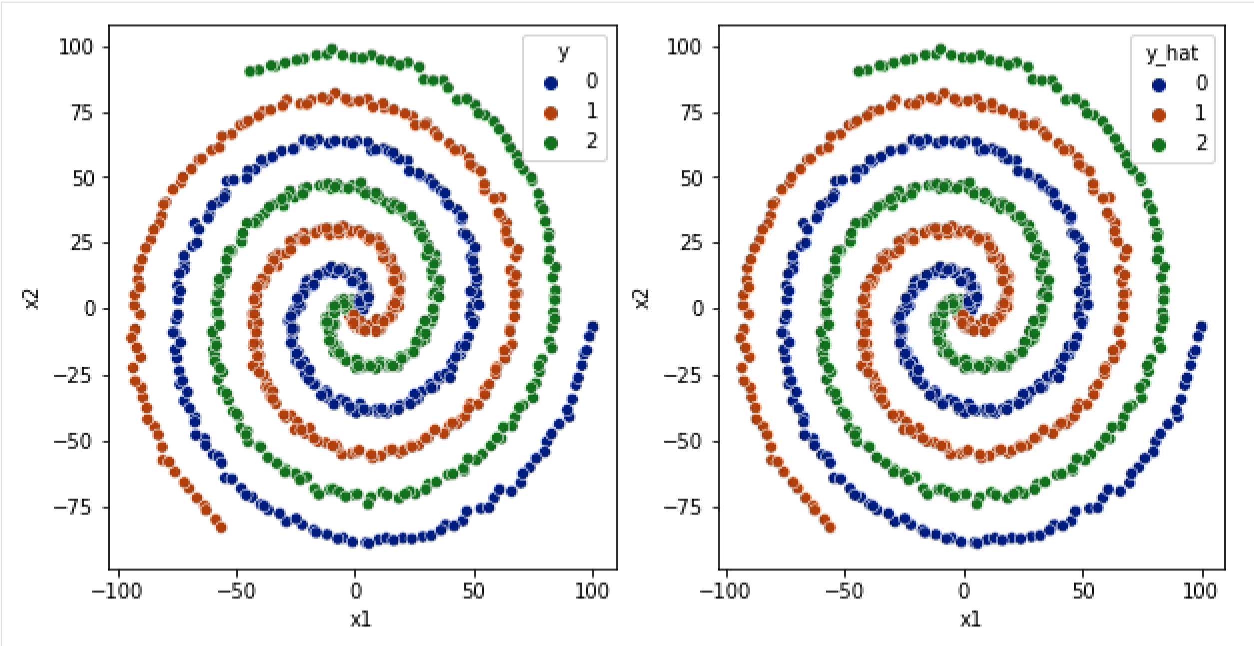
```
    loss=tf.keras.losses.CategoricalCrossentropy(),
    metrics=tf.keras.metrics.CategoricalAccuracy()
)

history = model.fit(
    df[['x1','x2']],
    y_ohe,
    epochs=500,
    batch_size=32,
    verbose=0,
    validation_split = 0.3
)

history_metrics = pd.DataFrame(history.history)
history_metrics['epochs'] = history.epoch
```

[54]:
```
fig,ax = plt.subplots(1,2,figsize=(10,5))

history_metrics.plot(x='epochs',y=['loss','val_loss'], ax=ax[0])
history_metrics.plot(x='epochs',y=['categorical_accuracy','val_categorical_accuracy'],␣
↪ax=ax[1])
```

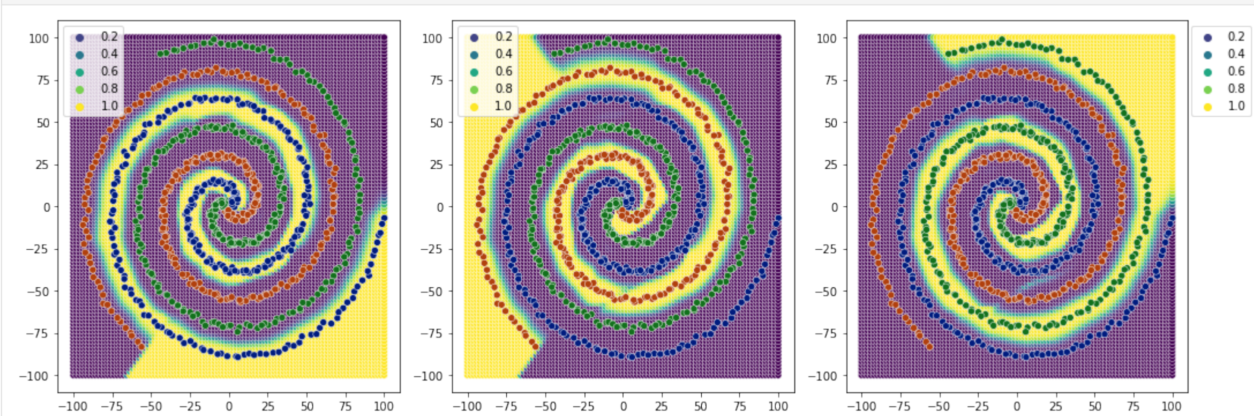
[54]:
```
<AxesSubplot:xlabel='epochs'>
```



[55]:
```
df['y_hat'] = np.argmax(model.predict(df[['x1','x2']].values), axis=1)

fig,ax = plt.subplots(1,2,figsize=(10,5))
sns.scatterplot(data=df,x='x1',y='x2',hue='y',ax=ax[0], palette='dark')
sns.scatterplot(data=df,x='x1',y='x2',hue='y_hat', ax=ax[1], palette='dark')
plt.show()
```



[56]:
```
plot_classification_boundary(model.predict,data=df[['x1','x2','y']].values,size=data_
↪limit,\
                                          figsize=(15,5), bound_details=100, n_plot_cols=3)
```



Clear decision boundaries.

11.3.3 A little bit complex model

[57]:
```
model = tf.keras.Sequential([
    tf.keras.layers.Dense(units=100, activation='relu'),
    tf.keras.layers.Dense(units=100, activation='relu'),
    tf.keras.layers.Dense(units=100, activation='relu'),
    tf.keras.layers.Dense(units=100, activation='relu'),
    tf.keras.layers.Dense(units=3, activation='softmax')
])
```

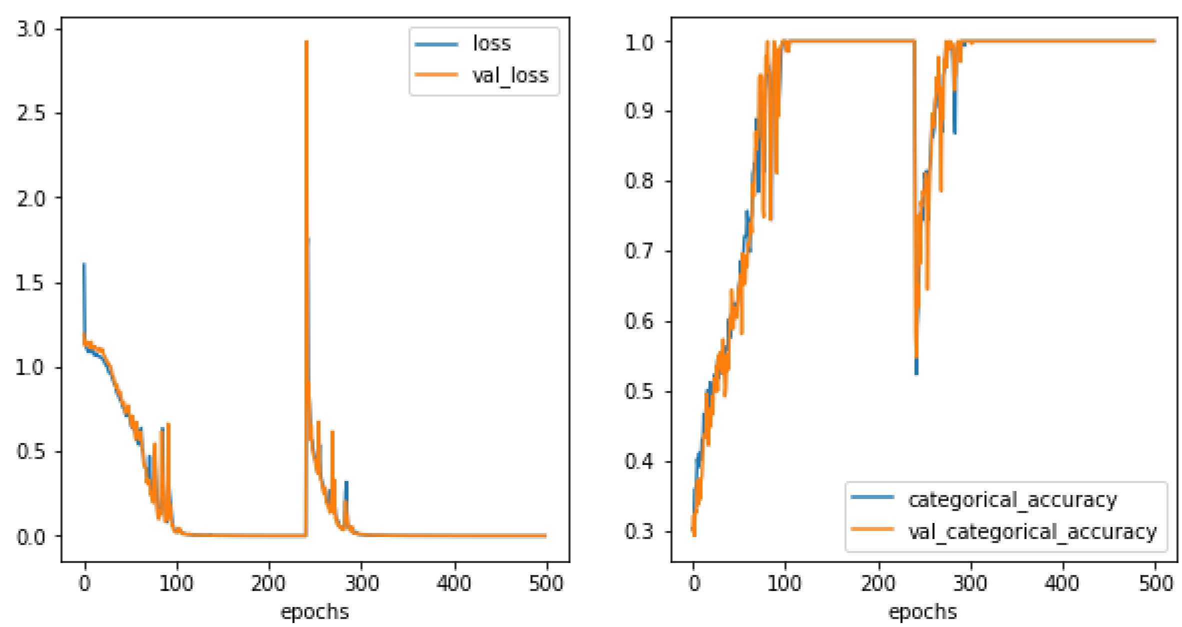
(continues on next page)

(continued from previous page)

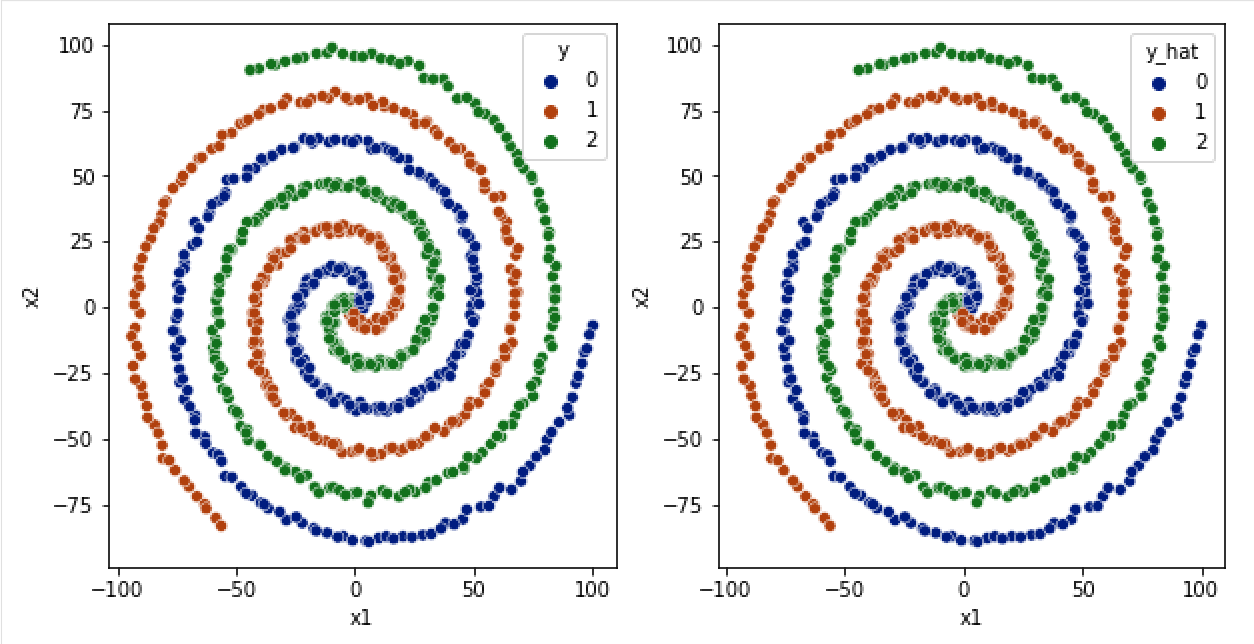
```
model.compile(
    optimizer=tf.optimizers.Adam(learning_rate=0.001),
    loss=tf.keras.losses.CategoricalCrossentropy(),
    metrics=tf.keras.metrics.CategoricalAccuracy()
)

history = model.fit(
    df[['x1','x2']],
    y_ohe,
    epochs=500,
    batch_size=32,
    verbose=0,
    validation_split = 0.3
)

history_metrics = pd.DataFrame(history.history)
history_metrics['epochs'] = history.epoch
```

[58]:
```
fig,ax = plt.subplots(1,2,figsize=(10,5))

history_metrics.plot(x='epochs',y=['loss','val_loss'], ax=ax[0])
history_metrics.plot(x='epochs',y=['categorical_accuracy','val_categorical_accuracy'],␣
↪ax=ax[1])
```

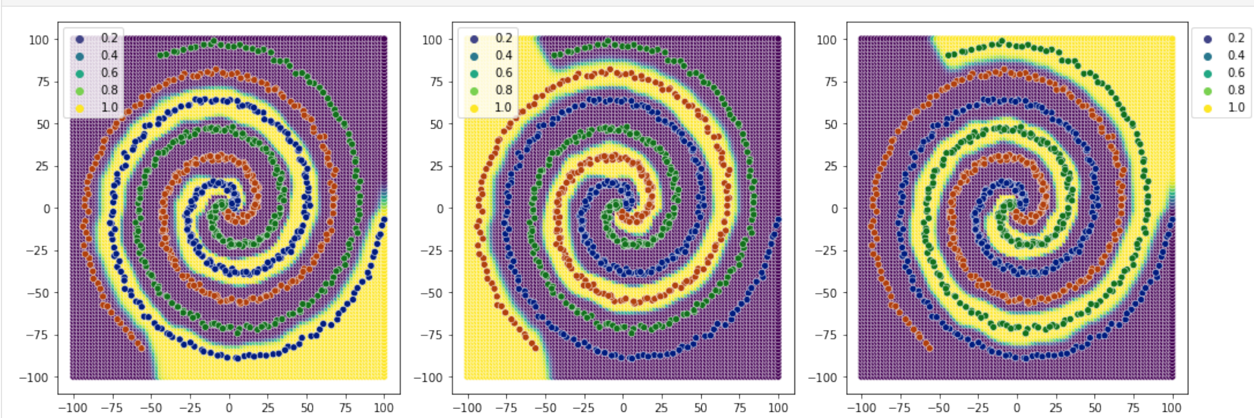
[58]:
```
<AxesSubplot:xlabel='epochs'>
```



[59]:
```
df['y_hat'] = np.argmax(model.predict(df[['x1','x2']].values), axis=1)

fig,ax = plt.subplots(1,2,figsize=(10,5))
sns.scatterplot(data=df,x='x1',y='x2',hue='y',ax=ax[0], palette='dark')
sns.scatterplot(data=df,x='x1',y='x2',hue='y_hat', ax=ax[1], palette='dark')
plt.show()
```



[60]:
```
plot_classification_boundary(model.predict,data=df[['x1','x2','y']].values,size=data_
↪limit,\
                                    figsize=(15,5), bound_details=100, n_plot_cols=3)
```



clearer decision boundaries than 2 relu layers.

CHAPTER

TWELVE

BASIC AUTOENCODERS

```
                              +----+
                              |    |
                              | C  |
      +-----------------+     | O  |        +------------------+
-->   |     Encoder     | --> | D  |   -->  |      Decoder     | -->
      +-----------------+     | E  |        +------------------+
                              |    |
                              |    |
                              |    |
                              +----+
```

[18]:
```
import warnings

warnings.filterwarnings('ignore')
```

[19]:
```
import numpy as np
from tensorflow import keras
import matplotlib.pyplot as plt
```

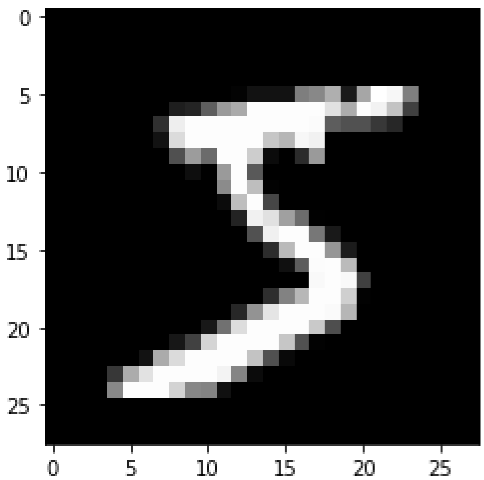
[20]:
```
(X_train, y_train), (X_test, y_test) = keras.datasets.mnist.load_data()
```

[21]:
```
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

[21]:
```
((60000, 28, 28), (10000, 28, 28), (60000,), (10000,))
```

[22]:
```
y_train[0], plt.imshow(X_train[0], cmap='gray')
```

[22]:
```
(5, <matplotlib.image.AxesImage at 0x7fe9d81fb760>)
```



[23]:
```
X_train = X_train/255
X_test = X_test/255
```

12.1 Architecture

[34]:
```
INPUT_SHAPE = (28, 28, 1)
```

[35]:
```
encoder_input = keras.layers.Input(shape=INPUT_SHAPE, name='input-layer')
flatten_layer = keras.layers.Flatten()(encoder_input)
encoder_output = keras.layers.Dense(units=64, activation="relu", name='encoder')(flatten_
↪layer)
hidden_layer2 = keras.layers.Dense(units=28*28, activation="relu", name='hidden-layer2
↪')(encoder_output)
decoder_output = keras.layers.Reshape(target_shape=INPUT_SHAPE, name='decoder')(hidden_
↪layer2)
```

[36]:
```
encoder = keras.Model(inputs=encoder_input, outputs=encoder_output)
auto_encoder = keras.Model(inputs=encoder_input, outputs=decoder_output)
```

[37]:
```
auto_encoder.summary()
```

```
Model: "model_5"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input-layer (InputLayer)    [(None, 28, 28, 1)]       0

 flatten_2 (Flatten)         (None, 784)               0

 encoder (Dense)             (None, 64)                50240

 hidden-layer2 (Dense)       (None, 784)               50960

```

(continues on next page)

(continued from previous page)

```
 decoder (Reshape)           (None, 28, 28, 1)         0

=================================================================
Total params: 101,200
Trainable params: 101,200
Non-trainable params: 0
_________________________________________________________________
```

[38]:
```
opt = keras.optimizers.Adam(learning_rate=0.001)

auto_encoder.compile(opt, loss='mse')
```

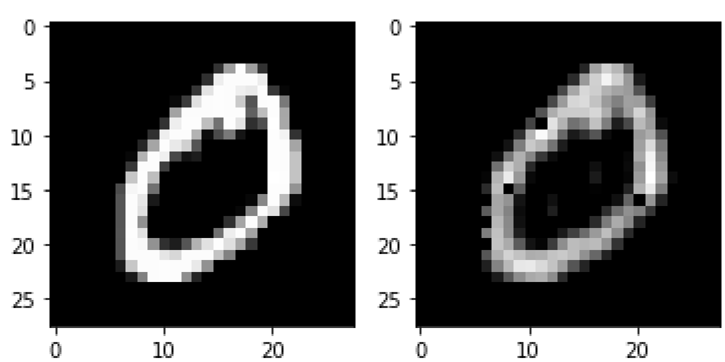
[39]:
```
epochs=3

for epoch in range(epochs):

    history = auto_encoder.fit(
        X_train,
        X_train,
        epochs=1,
        batch_size=32, validation_split=0.10)
```

```
1688/1688 [==============================] - 10s 5ms/step - loss: 0.0170 - val_loss: 0.
↪0115
1688/1688 [==============================] - 9s 5ms/step - loss: 0.0110 - val_loss: 0.
↪0107
1688/1688 [==============================] - 9s 5ms/step - loss: 0.0105 - val_loss: 0.
↪0104
```

[40]:
```
def plot_ae_images(arr):
    fig, ax = plt.subplots(1, 2)
    ax[0].imshow(arr, cmap='gray')
    ax[1].imshow(auto_encoder.predict(arr.reshape(-1, 28, 28, 1))[0], cmap='gray')
    plt.show()
```

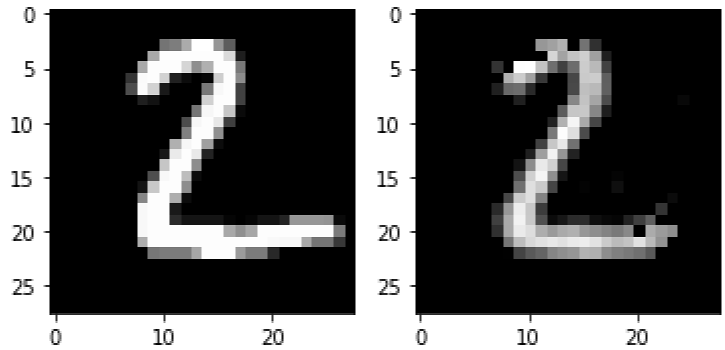
[41]:
```
plot_ae_images(X_train[1])
```

```
1/1 [==============================] - 1s 888ms/step
```



[42]:
```
plot_ae_images(X_test[1])
```

```
1/1 [==============================] - 0s 46ms/step
```



CHAPTER

THIRTEEN

INDICES AND TABLES

- genindex
- modindex
- search

INDEX